JUNOScript™ API Guide

Release 5.6

Juniper Networks, Inc.
1194 North Mathilda Avenue
Sunnyvale, CA 94089
USA
408-745-2000
www.juniper.net

Part Number: 530-008948-01, Revision 1

JUNOScript API Guide, Release 5.6
Copyright © 2002, Juniper Networks, Inc.
All rights reserved. Printed in USA.

Writer: Tony Mauro
Editor: Sonia Saruba
Covers and template design: Edmonds Design

Revision History
27 December 2002—First edition.

Table of Contents

List of Tables

List of Tables

About This Manual

This chapter provides a high-level overview of the *JUNOScript API Guide*:

- Objectives on page ix
- Audience on page x
- Document Organization on page x
- General Document Conventions on page xi
- List of Technical Publications on page xii
- Documentation Feedback on page xiii
- How to Request Support on page xiii

Objectives

This manual describes how to use the JUNOScript application programming interface (API) to configure or request information from the JUNOScript server running on a Juniper Networks router running JUNOS 5.6 Internet software. The JUNOScript API is a set of Extensible Markup Language (XML) tags that describe hardware and software installed and configured on the router.

This manual documents a specific release of the JUNOScript API, as indicated on the document title page. To obtain additional information about the JUNOScript API—either corrections to or information omitted from this manual—refer to the printed software release notes.

To obtain the most current version of this manual and the most current version of the software release notes, refer to the product documentation page on the Juniper Networks Web site, which is located at http://www.juniper.net/.

To order printed copies of this manual or to order a documentation CD-ROM, which contains this manual, please contact your sales representative.

Audience

This manual is designed for Juniper Networks customers who want to write custom applications for router configuration or monitoring. It assumes that you are familiar with basic terminology and concepts of XML and of XML-parsing utilities such as the Document Object Model (DOM) or Simple API for XML (SAX).

Document Organization

This manual contains the following parts and chapters:

- Preface, “About This Manual” (this chapter), provides a brief description of the contents and organization of this manual and describes how to obtain customer support.
- Part 1, “Overview,” provides an introduction to the JUNOScript API:
    - Chapter 1, “Introduction to the JUNOScript API,” briefly describes the JUNOScript API and XML, and outlines a communications session between the JUNOScript server and a client application.
- Part 2, “Session Control, Operational Requests, and Router Configuration,” describes how to use the JUNOScript API to monitor router status and to read or change router configuration:
    - Chapter 2, “JUNOScript Session Control,” explains how to start, control, and terminate a session with the JUNOScript server running on a Juniper Networks router, and describes the conventions that client applications must obey when exchanging JUNOScript-tagged data with the JUNOScript server.
    - Chapter 3, “Operational Requests,” explains how to use the JUNOScript API to request information about router status—the kind of information provided by operational mode commands in the JUNOS command-line interface (CLI).
    - Chapter 4, “Router Configuration,” describes how to use the JUNOScript API to change router configuration or to request information about the current configuration.
- Part 3, “Write JUNOScript Client Applications,” describes how to write client applications that automate access to the JUNOScript server and parse its output.
    - Chapter 5, “Write Perl Client Applications,” describes how to use the JUNOScript Perl module to speed and simplify implementation of client applications written in Perl. It includes a tutorial that uses sample Perl scripts to illustrate how to use the Perl module.
    - Chapter 6, “Write a C Client Application,” illustrates how a C-language client application connects to the JUNOScript server.

This manual also contains an index.

General Document Conventions

This document uses the following text conventions:

- Names of commands, files, and directories are shown in a sans serif font, as are configuration hierarchy levels. The following example refers to the ssh command:

  The client application invokes the ssh command.

- Examples of command output and the contents of files or XML document type definitions (DTDs) are shown in a fixed-width font when it is important to preserve the column alignment, or in sans serif font otherwise. The following example using sans serif font is from the junos-chassis DTD:

  ```
  <!ELEMENT chassis-inventory (chassis?)>
  ```

- Options, which are variable terms for which you substitute appropriate values, are shown in italics. The following example refers to the variable called *identifier*:

  For *identifier*, substitute the name of this instance of the object.

- XML tag names are shown in sans serif font and enclosed in angle brackets, which is the standard XML notation for tags. The angle brackets do not indicate that an element is optional, as they do in the syntax statement for a JUNOS CLI command. The following example refers to tags called <generation> and <local-index>:

  Notice that the JUNOScript server emits the <generation> tag before the <local-index> tag.

- Within a stream of XML tags, XML comments are enclosed within the strings <!-- and -->. The following example includes an XML comment:

  ```
  <configuration>
    <forwarding-options>
      <sampling>
        <disable/>
        <!-- other children of the <sampling> tag -->
      </sampling>
    </forwarding-options>
  </configuration>
  ```

  Outside an XML data set, an ellipsis (...) represents parts of a file or example that are omitted to highlight the remaining elements.

List of Technical Publications

Table 1 lists the software and hardware books for Juniper Networks routers and describes the contents of each book.

Table 1: Juniper Networks Technical Documentation

Book	Description
JUNOS Internet Software Configuration Guides	
Getting Started	Provides an overview of the JUNOS Internet software and describes how to install and upgrade the software. This manual also describes how to configure system management functions and how to configure the chassis, including user accounts, passwords, and redundancy.
Interfaces and Class of Service	Provides an overview of the interface and class-of-service functions of the JUNOS Internet software and describes how to configure the interfaces on the router.
MPLS Applications	Provides an overview of traffic engineering concepts and describes how to configure traffic engineering protocols.
Multicast	Provides an overview of multicast concepts and describes how to configure multicast routing protocols.
Network Management	Provides an overview of network management concepts and describes how to configure various network management features, such as SNMP, accounting options, and cflowd.
Policy Framework	Provides an overview of policy concepts and describes how to configure routing policy, firewall filters, and forwarding options.
Routing and Routing Protocols	Provides an overview of routing concepts and describes how to configure routing, routing instances, and unicast routing protocols.
VPNs	Provides an overview of Layer 2 and Layer 3 Virtual Private Networks (VPNs), describes how to configure VPNs, and provides configuration examples.
JUNOS Internet Software References	
Operational Mode Command Reference: Interfaces	Describes the JUNOS Internet software operational mode commands you use to monitor and troubleshoot interfaces on Juniper Networks M-series and T-series routers.
Operational Mode Command Reference: Protocols, Class of Service, Chassis, and Management	Describes the JUNOS Internet software operational mode commands you use to monitor and troubleshoot most aspects of Juniper Networks M-series and T-series routers.
System Log Messages Reference	Describes how to access and interpret system log messages generated by JUNOS software modules and provides a reference page for each message.
JUNOScript API Documentation	
JUNOScript API Guide	Describes how to use the JUNOScript API to monitor and configure Juniper Networks routers.
JUNOScript API Reference	Provides a reference page for each tag in the JUNOScript API.
JUNOS Internet Software Comprehensive Index	
Comprehensive Index	Provides a complete index of all JUNOS Internet software books and the *JUNOScript API Guide*.
Hardware Documentation	
Hardware Guide	Describes how to install, maintain, and troubleshoot routers and router components. Each router platform (M5 and M10 routers, M20 router, M40 router, M40e router, M160 router, T320 router, and T640 routing node) has its own hardware guide.
PIC Guide	Describes the router Physical Interface Cards (PICs). Each router platform has its own PIC guide.

Documentation Feedback

We are always interested in hearing from our customers. Please let us know what you like and do not like about the Juniper Networks documentation, and let us know of any suggestions you have for improving the documentation. Also, let us know if you find any mistakes in the documentation. Send your feedback to techpubs-comments@juniper.net.

How to Request Support

For technical support, contact Juniper Networks at support@juniper.net, or at 1-888-314-JTAC (within the United States) or 408-745-2121 (from outside the United States).

Part 1
Overview

Chapter 1
Introduction to the JUNOScript API

The JUNOScript application programming interface (API) is an Extensible Markup Language (XML) application that Juniper Networks routers use to exchange information with client applications. XML is a metalanguage for defining how to mark the organizational structures and individual items in a data set or document with tags that describe the function of the structures and items. The JUNOScript API defines tags for describing the components and configuration of routers.

Client applications can configure or request information from a router by encoding the request with JUNOScript tags and sending it to the JUNOScript server on the router. (The JUNOScript server is a component of the management daemon [mgd process] running on the router and does not appear as a separate entry in process listings.) The JUNOScript server directs the request to the appropriate software modules within the router, encodes the response in JUNOScript tags or formatted ASCII as requested by the client application, and returns the result to the client application. For example, to request information about the status of a router's interfaces, a client application can send the JUNOScript <get-interface-information> tag element. The JUNOScript server gathers the information and returns it in the <interface-information> tag element.

This manual explains how to use the JUNOScript API to configure Juniper Networks routers or request information about configuration or operation. The main focus is on writing client applications to interact with the JUNOScript server, but you can also use the JUNOScript API to build custom end-user interfaces for configuration and information retrieval and display, such as a Web browser-based interface.

This chapter discusses the following topics:

- About XML on page 4
- Advantages of Using the JUNOScript API on page 5
- Overview of a JUNOScript Session on page 6

About XML

XML is a language for defining a set of markers, called tags, that define the function and hierarchical relationships of the parts of a document or data set. The tags look much like Hypertext Markup Language (HTML) tags, but XML is actually a metalanguage used to define tags that best suit the kind of data being marked.

The following sections discuss XML and JUNOScript:

- XML and JUNOScript Tags on page 4
- Document Type Definition on page 5

For more details about XML, see *A Technical Introduction to XML* at http://www.xml.com/pub/98/10/guide0.html and the additional reference material at the xml.com site. The official XML specification is available at http://www.w3.org/TR/REC-xml.

XML and JUNOScript Tags

JUNOScript tags obey the XML convention that a tag name indicates the kind of information enclosed by the tag element. For example, the name of the JUNOScript <interface-state> tag element indicates that it contains a description of a router interface's current status, whereas the name of the <input-bytes> tag element indicates that its contents specify the number of bytes received.

JUNOScript tag names are enclosed in angle brackets, which is an XML convention. The brackets are a required part of the complete tag name, and are not to be confused with the angle brackets used in the JUNOS Internet software manuals to indicate optional parts of command-line interface (CLI) command strings.

When tagging items in an XML-compliant document or data set, you always enclose the item in paired opening and closing tags. XML is stricter in this respect than HTML, which sometimes uses only opening tags. The following examples show paired opening and closing tags:

```
<interface-state>enabled</interface-state>
<input-bytes>25378</input-bytes>
```

If a tag element is *empty*—has no contents—you can represent it either as a pair of opening and closing tags with nothing between them or as a single tag with a forward slash after the tag name. For example, the string <snmp-trap-flag/> represents the same empty tag as <snmp-trap-flag></snmp-trap-flag>.

When discussing tags in text, this manual conventionally uses just the name of the opening tag to represent the complete tag element (opening tag, contents, and closing tag). For example, it usually refers to "the <input-bytes> tag element" rather than "the <input-bytes>*number-of-bytes*</input-bytes> tag element."

Document Type Definition

An XML-tagged document or data set is *structured*, because a set of rules specifies the ordering and interrelationships of the items in it. The rules define the contexts in which each tagged item can—and in some cases must—occur. A file called a *document type definition*, or *DTD*, lists every tag element that can appear in the document or data set, defines the parent-child relationships between the tags, and specifies other tag characteristics. The same DTD can apply to many XML documents or data sets.

Advantages of Using the JUNOScript API

The JUNOScript API is a programmatic interface. The JUNOScript DTDs fully document all options for every command and all elements in a configuration statement. JUNOScript tag names clearly indicate the function of an element in a command or configuration statement.

The combination of meaningful tag names and the structural rules in a DTD makes it easy to understand the content and structure of an XML-tagged data set or document. JUNOScript tags make it straightforward for client applications that request information from a router to parse the output and find specific information.

The following example illustrates how the JUNOScript API makes it easier to parse router output and extract the needed information. It compares formatted ASCII and XML-tagged versions of output from a router. The formatted ASCII follows:

```
Physical interface: fxp0, Enabled, Physical link is Up
  Interface index: 4, SNMP ifIndex: 3
```

This is the JUNOScript-tagged version:

```
<interface>
  <name>fxp0</name>
  <admin-status>enabled</admin-status>
  <operational-status>up</operational-status>
  <index>4</index>
  <snmp-index>3</snmp-index>
</interface>
```

When a client application needs to extract a specific piece of data from formatted ASCII output, it must rely on the datum’s location, expressed either absolutely or with respect to adjacent strings. Suppose that the client application wants to extract the interface index. It can use a utility such as `expect` to locate specific strings, but one difficulty is that the number of digits in the interface index is not necessarily predictable. The client application cannot simply read a certain number of characters after the `Interface index`: label, but must instead extract everything between the label and the subsequent string:

```
, SNMP ifIndex
```

A problem arises if the format or ordering of output changes in a later version of the software, for example, if a `Logical index` field is added following the interface index number:

```
Physical interface: fxp0, Enabled, Physical link is Up
  Interface index: 4, Logical index: 12, SNMP ifIndex: 3
```

A search for the interface index number that relies on the SNMP ifIndex string now returns an incorrect result. The client application must be updated manually to search for the following string instead:

```
, Logical index
```

In contrast, the structured nature of the JUNOScript-tagged output enables a client application to retrieve the interface index by extracting everything within the opening <index> tag and closing </index> tag. The application does not have to rely on an element's position in the output string, so the JUNOScript server can emit the child tags in any order within the <interface> tags. Adding a new <logical-index> tag element in a future release does not affect an application's ability to locate the <index> tag element and extract its contents.

Tagged output is also easier to transform into different display formats. For instance, you might want to display different amounts of detail about a given router component at different times. When a router returns formatted ASCII output, you have to design and write special routines and data structures in your display program to extract and store the information needed for a given detail level. In contrast, the inherent structure of JUNOScript output is an ideal basis for a display program's own structures. It is also easy to use the same extraction routine for several levels of detail, simply ignoring the tags you do not need when creating a less detailed display.

Overview of a JUNOScript Session

Communication between the JUNOScript server and a client application is session-based: the two parties explicitly establish a connection before exchanging data and close the connection when they are finished. The streams of JUNOScript tags emitted by the JUNOScript server and a client application each constitute a *well-formed* XML document, because the tag streams obey the structural rules defined in the JUNOScript DTDs for the kind of information they encode. Client applications must produce a well-formed XML document by emitting tags in the required order and only in the legal contexts.

The following list outlines the basic structure of a JUNOScript session. For more specific information, see “Start, Control, and End a JUNOScript Session” on page 14.

1. The client application establishes a connection to the JUNOScript server and opens the JUNOScript session.

2. The JUNOScript server and client application exchange initialization tags, used to determine if they are using compatible versions of the JUNOS Internet software and the JUNOScript API.

3. The client application sends one or more requests to the JUNOScript server and parses its responses.

4. The client application closes the JUNOScript session and the connection to the JUNOScript server.

Part 2
Session Control, Operational Requests, and Router Configuration

Chapter 2
JUNOScript Session Control

This chapter explains how to start and terminate a session with the JUNOScript server, and describes the Extensible Markup Language (XML) tag elements that client applications and the JUNOScript server use to coordinate information exchange during the session. It discusses the following topics:

- General JUNOScript Conventions on page 9
- Start, Control, and End a JUNOScript Session on page 14
- Handle an Error Condition on page 32
- Halt a Request on page 32
- Display CLI Output as JUNOScript Tag Elements on page 33
- Example of a JUNOScript Session on page 33

General JUNOScript Conventions

A client application must comply with XML and JUNOScript conventions. Compliant applications are easier to maintain if the JUNOS Internet software or the JUNOScript application programming interface (API) changes. The JUNOScript server always obeys the conventions. The following sections describe JUNOScript conventions:

- Ordering and Context for Session Control Tag Elements on page 10
- Ordering and Context for Request and Response Tag Elements on page 10
- Ordering and Context for a Request Tag Element's Child Tag Elements on page 11
- Ordering and Context for a Response Tag Element Element's Child Tag Elements on page 11
- Spaces, Newlines, and Other White Space Characters on page 11
- XML Comments on page 12
- XML Processing Instructions on page 12
- Predefined Entity References on page 12

Ordering and Context for Session Control Tag Elements

A *session control* tag element is one that delimits the parts of a JUNOScript session. There are tag elements that indicate the start or end of a session, identify a client request or server response, and signal error conditions. Most session control tag elements can occur only in certain contexts and in a prescribed order. JUNOScript session controllers include the following:

- <?xml?>—An XML processing instruction (PI), emitted by both the client application and the JUNOScript server as they establish a JUNOScript session. For more information about PIs, see “XML Processing Instructions” on page 12.
- <junoscript>—The root tag element for every JUNOScript session, emitted by both the client application and the JUNOScript server as they establish and close the session.
- <rpc>—The container tag element that encloses each request emitted by the client application. It can occur only within the <junoscript> tag element.
- <rpc-reply>—The container tag element that encloses each response returned by the JUNOScript server. It can occur only within the <junoscript> tag element.

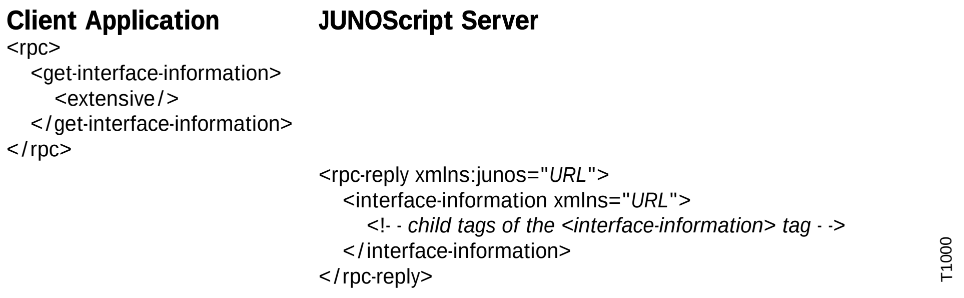
For more information about how to use these tag elements, see “Start, Control, and End a JUNOScript Session” on page 14 and the summary of session control tag elements in the *JUNOScript API Reference*.

Ordering and Context for Request and Response Tag Elements

A *request* tag element is one generated by a client application to request information about a router’s current status or configuration or to change the configuration. A request tag element corresponds to a JUNOS command-line interface (CLI) operational command or configuration statement. It can occur only within an <rpc> session control tag element.

A *response* tag element represents the JUNOScript server’s reply to a request tag element and occurs only within an <rpc-reply> tag element.

The following example represents an exchange in which a client application emits the <get-interface-information> request tag element with the <extensive/> flag and the JUNOScript server returns the <interface-information> response tag element. (For information about the xmlns:junos and xmlns attributes, see “Parse the JUNOScript Server Response” on page 28.)



Client Application **JUNOScript Server**

```
<rpc>
   <get-interface-information>
      <extensive/>
   </get-interface-information>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
   <interface-information xmlns="URL">
      <!- - child tags of the <interface-information> tag - ->
   </interface-information>
</rpc-reply>
```

T1000

Note

A client application can send only one request tag element at a time to a particular router, and must not send another request tag element until it receives the closing </rpc-reply> tag that represents the end of the JUNOScript server's response to the current request.

Ordering and Context for a Request Tag Element's Child Tag Elements

Some request tag elements contain child tag elements. For configuration request tag elements, each child tag element represents a level in the JUNOS configuration hierarchy. For operational request tag elements, each child tag element represents one of the options you provide on the command line when issuing the equivalent CLI command.

Some request tag elements require that certain child tag elements be present. To make a request successfully, a client application must emit the required child tag elements within the request tag element's opening and closing tags. If any of the request tag element's children are themselves container tag elements, the opening tag for each must occur before any of the tag elements it contains, and the closing tag must occur before the opening tag for another tag element at its hierarchy level.

In most cases, the ordering of child tag elements at one hierarchy level within a request tag element is not significant. The important exception is the *identifier tag* for a configuration element, which distinguishes the configuration element from other elements of its type. It must occur first within the container tag element that represents the configuration element. Most often the identifier tag element specifies the configuration element name and is called <name>. For more information, see "Tag Element Mappings for Identifiers" on page 44.

Ordering and Context for a Response Tag Element Element's Child Tag Elements

The child tag elements of a response tag element represent the individual data items returned by the JUNOScript server for a particular request. At one hierarchy level within a response tag element, there is no prescribed order for the child tag elements, and the set of child tag elements is subject to change in later releases of the JUNOScript API. Client applications must not rely on the presence or absence of a particular tag element in the JUNOScript server's output, nor on the ordering of child tag elements within a response tag element. For the most robust operation, include logic in the client application that handles the absence of expected tag elements or the presence of unexpected ones as gracefully as possible.

Spaces, Newlines, and Other White Space Characters

The JUNOScript API complies with the XML specification in ignoring spaces, newlines, and other characters that represent white space. Client applications must not depend on leading, trailing, or embedded white space when parsing the tag stream emitted by the JUNOScript server. For more information about white space in XML documents, see the XML specification at http://www.w3.org/TR/REC-xml.

XML Comments

XML comments can appear at any point in the tag stream emitted by the JUNOScript server. Client applications must handle them gracefully but must not depend on their content. Client applications also cannot use comments to convey information to the JUNOScript server, because the server automatically discards any comments it receives.

XML comments are enclosed within the strings `<!--` and `-->`, and cannot contain the string `--` (two hyphens). For more details about comments, see the XML specification at http://www.w3.org/TR/REC-xml.

The following is an example of an XML comment:

```
<!- This is a comment. Please ignore it. -->
```

XML Processing Instructions

An XML PI contains information relevant to a particular protocol and has the following form:

<?*PI-name attributes*?>

Some PIs emitted during a JUNOScript session include information that a client application needs for correct operation. A prominent example is the `<?xml?>` tag element, which the client application and JUNOScript server each emit at the beginning of every JUNOScript session to specify which version of XML and which character encoding scheme they are using. For more information, see “Emit the Initialization PI and Tag” on page 20.

The JUNOScript server can also emit PIs that the client application does not need to interpret (for example, PIs intended for the JUNOS CLI). If the client application does not understand a PI, it must treat the PI like a comment instead of exiting or generating an error message.

Predefined Entity References

By XML convention, there are two contexts in which certain characters cannot appear in their regular form:

- In the string that appears between opening and closing tags (the contents of the tag element)
- In the string value assigned to an attribute of an opening tag

When including a disallowed character in either context, client applications must substitute the equivalent *predefined entity reference*, which is a string of characters that represents the disallowed character. Because the JUNOScript server uses the same predefined entity references in its response tag elements, the client application must be able to convert them to actual characters when processing response tag elements.

Table 2 summarizes the mapping between disallowed characters and predefined entity references for strings that appear between the opening and closing tags of a tag element.

Table 2: Predefined Entity Reference Substitutions for Tag Content Values

Disallowed Character	Predefined Entity Reference
& (ampersand)	&amp;
> (greater-than sign)	&gt;
< (less-than sign)	&lt;

Table 3 summarizes the mapping between disallowed characters and predefined entity references for attribute values.

Table 3: Predefined Entity Reference Substitutions for Attribute Values

Disallowed Character	Predefined Entity Reference
& (ampersand)	&amp;
' (apostrophe)	&apos;
> (greater-than sign)	&gt;
< (less-than sign)	&lt;
" (quotation mark)	&quot;

As an example, suppose that the following string is the value contained by the <condition> tag element:

```
if (a<b && b>c) return "Peer's not responding"
```

Using the required predefined entity references, the <condition> tag element looks like this:

```
<condition>if (a&lt;b &amp;&amp; b&gt;c) return "Peer's not responding"</condition>
```

Similarly, if the value for the <example> tag element's heading attribute is Peer's "age" <> 40, the opening tag looks like this when the required predefined entity references are used:

```
<example heading="Peer&apos;s &quot;age&quot; &lt;&gt; 40">
```

Start, Control, and End a JUNOScript Session

The JUNOScript server communicates with client applications within the context of a JUNOScript *session*. The server and client explicitly establish a connection and session before exchanging data, and close the session and connection when they are finished. The streams of JUNOScript tag elements emitted by the JUNOScript server and a client application must each constitute a well-formed XML document by obeying the structural rules defined in the JUNOScript document type definition (DTD) for the kind of information they are exchanging. The client application must emit tag elements in the required order and only in the allowed contexts.

Client applications access the JUNOScript server using one of the protocols listed in "Supported Access Protocols" on page 14. To authenticate with the JUNOScript server, they use either a JUNOScript-specific mechanism or the protocol's standard authentication mechanism, depending on the protocol. After authentication, the JUNOScript server uses the JUNOS login accounts and classes already configured on the router to determine whether a client application is authorized to make each request.

See the following sections for information about establishing, using, and terminating a connection and JUNOScript session:

- Supported Access Protocols on page 14
- Prerequisites for Establishing a Connection on page 15
- Connect to the JUNOScript Server on page 18
- Start the JUNOScript Session on page 19
- Authenticate with the JUNOScript Server on page 24
- Exchange Tagged Data on page 26
- End the Session and Close the Connection on page 31

For an example of a complete JUNOScript session, see "Example of a JUNOScript Session" on page 33.

Supported Access Protocols

The JUNOScript server accepts connections created using the access protocols listed in Table 4, which also specifies the associated authentication mechanism.

Table 4: Supported Access Protocols and Authentication Mechanisms

Access Protocol	Authentication Mechanism
clear-text, a JUNOScript-specific protocol for sending unencrypted text over a Transmission Control Protocol (TCP) connection	JUNOScript-specific
ssh (secure shell)	Standard ssh
SSL (Secure Sockets Layer)	JUNOScript-specific
telnet	Standard telnet

The SSL and ssh protocols are preferred because they encrypt security information (such as a password) before transmitting it across the network. The clear-text and telnet protocols do not encrypt security information.

For information about the authentication prerequisites for each protocol, see "Prerequisites for Establishing a Connection" on page 15. For authentication instructions, see "Authenticate with the JUNOScript Server" on page 24.

Prerequisites for Establishing a Connection

Both the JUNOScript server and the client application must be able to access the software for the access protocol that the client application uses to create a connection. The JUNOScript server can access the protocols listed in "Supported Access Protocols" on page 14, because the JUNOS Internet software distribution includes them. On most operating systems, client applications can access the software for TCP (used by the JUNOScript-specific clear-text protocol) and the telnet protocol as part of the standard distribution. For information about obtaining ssh software for use by a client application, see http://www.ssh.com and http://www.openssh.com. For information about obtaining SSL software, see http://www.openssl.org.

For information about connection prerequisites, see the following sections:

- Prerequisites for clear-text Connections on page 15
- Prerequisites for ssh Connections on page 16
- Prerequisites for SSL Connections on page 16
- Prerequisites for telnet Connections on page 17

Prerequisites for clear-text Connections

If the client application uses the clear-text protocol to send unencrypted text directly over a TCP connection without using any additional protocol (such as ssh, SSL, or telnet), perform the following procedure to activate the xnm-clear-text service on port 3221 on the JUNOScript server machine:

1. Enter CLI configuration mode on the JUNOScript server machine and issue the following command:

   ```
   [edit]
   user@host# set system services xnm-clear-text
   ```

2. Commit the configuration:

   ```
   [edit]
   user@host# commit
   ```

Prerequisites for ssh Connections

The ssh protocol uses public-private key technology. The ssh client software must be installed on the machine where the client application runs. If the ssh private key is encrypted (as is recommended for greater security), the ssh client must be able to access the passphrase used to decrypt the key.

If the client application uses the JUNOScript Perl module described in “Write Perl Client Applications” on page 81, no further action is necessary. As part of the Perl module installation procedure, you install a prerequisites package that includes the necessary ssh software.

If the client application does not use the JUNOScript Perl module, perform the following procedures to enable it to establish ssh connections:

1. Install the ssh client on the machine where the client application runs.
2. If the private key is encrypted (as recommended), use one of the following methods to make the associated passphrase available to the ssh client:
   - Run the ssh-agent program to provide key management.
   - Direct the ssh client to the file on the local disk that stores the passphrase.
   - Include code in the client application that prompts a user for the passphrase.

For more information about configuring these methods, see the ssh documentation.

Prerequisites for SSL Connections

The SSL protocol uses public-private key technology, which requires a paired private key and authentication certificate. Perform the following procedure to enable a client application to establish SSL connections:

1. Install the SSL client on the machine where the client application runs.

   (Skip this step if the client application uses the JUNOScript Perl module described in “Write Perl Client Applications” on page 81. As part of the Perl module installation procedure, you install a prerequisites package that includes the necessary SSL software.)

2. Obtain an authentication certificate in Privacy Enhanced Mail (PEM) format, in one of two ways:
   - Request a certificate from a Certificate Authority; these agencies usually charge a fee.
   - Issue the following openssl command to generate a self-signed certificate; for information about obtaining the openssl software, see http://www.openssl.org.

     The command writes the certificate and an unencrypted 1024-bit RSA private key to the file called *certificate-file*.pem. The command appears here on two lines only for legibility:

```
% openssl req -x509 -nodes -newkey rsa:1024 -keyout certificate-file.pem \
        -out certificate-file.pem
```

3. Enter CLI configuration mode on the JUNOScript server machine and issue the following commands to import the certificate. In the first command, substitute the desired certificate name for the *certificate-name* variable. In the second command, for the *URL-or-path* variable substitute the name of the file that contains the paired certificate and private key, either as a URL or a pathname on the local disk:

```
[edit]
user@host# edit security certificates local certificate-name

[edit security certificates local certificate-name]
user@host# set load-key-file URL-or-path
```

Note

The CLI expects the private key in the specified file (*URL-or-path*) to be unencrypted. If the key is encrypted, the CLI prompts for the passphrase associated with it, decrypts it, and stores the unencrypted version.

4. Issue the following commands to activate the xnm-ssl service, which listens on port 3220. In the last command, substitute the same value for the *certificate-name* variable as in Step 3:

```
[edit security certificates local certificate-name]
user@host# top

[edit]
user@host# edit system services

[edit system services]
user@host# activate xnm-ssl

[edit system services]
user@host# set xnm-ssl local-certificate certificate-name
```

5. Commit the configuration:

```
[edit system services]
user@host# commit
```

Prerequisites for telnet Connections

There are no prerequisites for enabling a client application to establish telnet connections, other than ensuring that both the client application and the JUNOScript server can access the telnet software. For a discussion, see “Prerequisites for Establishing a Connection” on page 15.

Connect to the JUNOScript Server

A client application written in Perl can most efficiently establish a connection and open a JUNOScript session by using the JUNOScript Perl module described in “Write Perl Client Applications” on page 81. For more information, see that chapter.

For a client application that does not use the JUNOScript Perl module, first perform the prerequisite procedures for the access protocol being used, as described in “Prerequisites for Establishing a Connection” on page 15. The supported access protocols are listed in “Supported Access Protocols” on page 14.

When the prerequisites are satisfied, the client application connects to the JUNOScript server by opening a socket or other communications channel to the JUNOScript server machine (router), invoking one of the remote-connection routines appropriate for the programming language and access protocol that the application uses.

What the client application does next depends on which access protocol it is using:

- If using the clear-text or SSL protocol, the client application does the following:

  1. Emits initialization tag elements, as described in “Start the JUNOScript Session” on page 19.

  2. Authenticates with the JUNOScript server, as described in “Authenticate with the JUNOScript Server” on page 24.

- If using the ssh or telnet protocol, the client application does the following:

  1. Uses the protocol’s built-in authentication mechanism to authenticate.

  2. Issues the junoscript command to request that the JUNOScript server convert the connection into a JUNOScript session. For a C programming language example, see “Write a C Client Application” on page 111.

     Note: Client applications that use the telnet protocol should add the no-echo argument to the junoscript command in order to prevent the JUNOScript server from echoing back data it receives from the application:

     ```
     junoscript no-echo
     ```

     Note: When using the telnet protocol to connect to a JUNOScript server on a router running JUNOS 5.4 and earlier, the JUNOS user account under which the client application runs must have the JUNOS shell permission bit (configured at the [edit system login class permissions] hierarchy level). If the JUNOScript server is running JUNOS 5.5 or later, the shell permission bit is not required.

  3. Emits initialization tag elements, as described in “Start the JUNOScript Session” on page 19.

Connect to the JUNOScript Server from the CLI

The JUNOScript API is primarily intended for use by client applications; however, for testing purposes you can establish an interactive JUNOScript session and type commands in a shell window. To connect to the JUNOScript server from JUNOS CLI operational mode, issue the junoscript command:

```
user@host> junoscript
```

To begin a JUNOScript session over the connection, emit the initialization PI and tag described in "Emit the Initialization PI and Tag" on page 20. You can then type sequences of tag elements that represent operational and configuration operations, as described in "Operational Requests" on page 37 and "Router Configuration" on page 41. To eliminate typing errors, save complete tag element sequences in a file and use a cut-and-paste utility to copy the sequences to the shell window.

Note

When you close the connection to the JUNOScript server (for example, by emitting the <request-end-session/> and </junoscript> tags), the router completely closes your connection instead of returning you to the CLI operational mode prompt. For more information about ending a JUNOScript session, see "End the Session and Close the Connection" on page 31.

Similarly, the JUNOScript server completely closes your connection if there are any typographical or syntax errors in the tag sequence you emit.

Start the JUNOScript Session

Each JUNOScript session begins with a handshake in which the JUNOScript server and the client application specify the versions of XML and the JUNOScript API they are using. Each party parses the version information emitted by the other, using it to determine whether they can communicate successfully. The following sections describe how to start a JUNOScript session:

- Emit the Initialization PI and Tag on page 20
- Parse the Initialization PI and Tag from the JUNOScript Server on page 21
- Verify Compatibility on page 23
- Supported Protocol Versions on page 24

Emit the Initialization PI and Tag

When the JUNOScript session begins, the client application emits an <?xml?> PI and an opening <junoscript> tag, as described in the following sections.

Emit the <?xml?> PI

The client application begins by emitting an <?xml?> PI with the following syntax:

```
<?xml version="version" encoding="encoding"?>
```

The PI attributes are as follows. For a list of the attribute values that are acceptable in the current version of the JUNOScript API, see "Supported Protocol Versions" on page 24.

- version—The version of XML with which tag elements emitted by the client application comply
- encoding—The standardized character set that the client application uses and can understand

In the following example of a client application's <?xml?> PI, the version="1.0" attribute indicates that it is emitting tag elements that comply with the XML 1.0 specification. The encoding="us-ascii" attribute indicates that the client application is using the 7-bit ASCII character set standardized by the American National Standards Institute (ANSI). For more information about ANSI standards, see http://www.ansi.org.

```
<?xml version="1.0" encoding="us-ascii"?>
```

Emit the Opening <junoscript> Tag

The client application then emits its opening <junoscript> tag, which has the following syntax:

```
<junoscript version="version" hostname="hostname" release="release-code">
```

The tag attributes are as follows. The version attribute is required, but the other attributes are optional. For a list of the attribute values that are acceptable in the current version of the JUNOScript API, see "Supported Protocol Versions" on page 24.

- version—(Required) The version of the JUNOScript API that the client application is using.
- hostname—The name of the machine on which the client application is running. The information is used only when diagnosing problems. The JUNOScript API does not include support for establishing trusted-host relationships or otherwise altering JUNOScript server behavior depending on the client hostname.

- release—The identifier of the JUNOS Internet software release for which the client application is designed. It indicates that the client application can interoperate successfully with a JUNOScript server designed to understand that version of the JUNOS Internet software. In other words, it indicates that the client application emits request tag elements corresponding to supported features of the indicated JUNOS Internet software version, and knows how to parse response tag elements that correspond to those features. If you do not include this attribute, the JUNOScript server assumes that the client application can interoperate with its version of the JUNOS Internet software. For more information, see "Verify Compatibility" on page 23.

  For *release-code*, use the standard notation for JUNOS Internet software version numbers. For example, the value 5.6R1 represents the initial version of JUNOS Release 5.6.

In the following example of a client application's opening <junoscript> tag, the version="1.0" attribute indicates that it is using JUNOScript version 1.0. The hostname="client1" attribute indicates that the client application is running on the machine called client1. The release="5.6R1" attribute indicates that the router is running the initial version of JUNOS Release 5.6.

```
<junoscript version="1.0" hostname="client1" release="5.6R1">
```

Parse the Initialization PI and Tag from the JUNOScript Server

When the JUNOScript session begins, the JUNOScript server emits an <?xml?> PI and an opening <junoscript> tag, as described in the following sections.

Parse the JUNOScript Server's <?xml?> PI

The syntax for the <?xml?> PI is as follows:

```
<?xml version="version" encoding="encoding"?>
```

The PI attributes are as follows. For a list of the attribute values that are acceptable in the current version of the JUNOScript API, see "Supported Protocol Versions" on page 24.

- version—The version of XML with which tag elements emitted by the JUNOScript server comply

- encoding—The standardized character set that the JUNOScript server uses and can understand

In the following example of a JUNOScript server's <?xml?> PI, the version="1.0" attribute indicates that the server is emitting tag elements that comply with the XML 1.0 specification. The encoding="us-ascii" attribute indicates that the server is using the 7-bit ASCII character set standardized by ANSI. For more information about ANSI standards, see http://www.ansi.org.

```
<?xml version="1.0" encoding="us-ascii"?>
```

Parse the JUNOScript Server's Opening <junoscript> Tag

The server then emits its opening <junoscript> tag, which has the following form (the tag appears on multiple lines only for legibility):

```
<junoscript version="version" hostname="hostname" os="JUNOS" release="release-code"
            xmlns="namespace-URL" xmlns:junos="namespace-URL"
            xmlns:xnm="namespace-URL">
```

The tag attributes are as follows.

- version—The version of the JUNOScript API that the JUNOScript server is using.
- hostname—The name of the router on which the JUNOScript server is running.
- os—The operating system of the router on which the JUNOScript server is running. The value is always JUNOS.
- release—The identifier for the version of the JUNOS Internet software from which the JUNOScript server is derived and is designed to understand. It is presumably in use on the router where the JUNOScript server is running. The *release-code* uses the standard notation for JUNOS Internet software version numbers. For example, the value 5.6R1 represents the initial version of JUNOS Release 5.6.
- xmlns—The XML namespace for the tag elements enclosed by the <junoscript> tag element that do not have a prefix on their names (that is, the default namespace for JUNOScript tag elements). The value is a URL of the form http://xml.juniper.net/xnm/*version*/xnm, where *version* is a string such as 1.1.
- xmlns:junos—The XML namespace for the tag elements enclosed by the <junoscript> tag element that have the junos: prefix on their names. The value is a URL of the form http://xml.juniper.net/junos/*release-code*/junos, where *release-code* is the standard string that represents a release of the JUNOS software. For example, the value 5.6R1 represents the initial version of JUNOS Release 5.6.
- xmlns:xnm—The XML namespace for the JUNOScript tag elements enclosed by the <junoscript> tag element that have the xnm: prefix on their names. The value is a URL of the form http://xml.juniper.net/xnm/*version*/xnm, where *version* is a string such as 1.1.

In the following example of a JUNOScript server's opening <junoscript> tag, the version attribute indicates that the server is using JUNOScript version 1.0 and the hostname attribute indicates that the router's name is big-router. The os and release attributes indicate that the router is running the initial version of JUNOS Release 5.6. The xmlns and xmlns:xnm attributes indicate that the default namespace for JUNOScript tag elements and the namespace for tag elements that have the xnm: prefix is http://xml.juniper.net/xnm/1.1/xnm. The xmlns:junos attribute indicates that the namespace for tag elements that have the junos: prefix is http://xml.juniper.net/junos/5.6R1/junos. The tag appears on multiple lines only for legibility.

```
<junoscript version="1.0" hostname="big-router" os="JUNOS" release="5.6R1"
            xmlns="http://xml.juniper.net/xnm/1.1/xnm"
            xmlns:junos="http://xml.juniper.net/junos/5.6R1/junos"
            xmlns:xnm="http://xml.juniper.net/xnm/1.1/xnm">
```

Verify Compatibility

Exchanging <?xml?> and <junoscript> tag elements enables a client application and the JUNOScript server to determine if they are running different versions of a protocol. Different versions are sometimes incompatible, and by JUNOScript convention the party running the later version of a protocol determines how to handle any incompatibility. For fully automated performance, include code in the client application that determines if its version of a protocol is later than that of the JUNOScript server. Decide which of the following options is appropriate when the application's version of a protocol is more recent, and implement the corresponding response:

- Ignore the version difference, and do not alter standard behavior to accommodate the JUNOScript server's version. A version difference does not always imply incompatibility, so this is often a valid response.
- Alter standard behavior to provide backward compatibility to the JUNOScript server. If the client application is running a later version of the JUNOS Internet software, for example, it can choose to emit only tag elements that represent the software features available in the JUNOScript server's version of the JUNOS Internet software.
- End the JUNOScript session and terminate the connection. This is appropriate if you decide that accommodating the JUNOScript server's version of a protocol is not practical. For instructions, see "End the Session and Close the Connection" on page 31.

Supported Protocol Versions

Table 5 lists the protocol versions supported by version 1.0 of the JUNOScript API and specifies the PI or opening tag and attribute used to specify the information during JUNOScript session initialization.

Table 5: Supported Protocol Versions

Protocol and Versions	PI or Tag	Attribute
XML 1.0	<?xml?>	version="1.0"
ANSI-standardized 7-bit ASCII character set	<?xml?>	encoding="us-ascii"
JUNOScript 1.0	<junoscript>	version="1.0"

JUNOS Release 5.4
JUNOS Release 5.5
JUNOS Release 5.6
	<junoscript>	release="5.4R*x*"
		release="5.5R*x*"
		release="5.6R*x*"

Authenticate with the JUNOScript Server

A client application that uses the clear-text or SSL protocol must now authenticate with the JUNOScript server. (Applications that use the ssh or telnet protocol use the protocol's built-in authentication mechanism before emitting initialization tag elements, as described in "Connect to the JUNOScript Server" on page 18.)

The client application that uses the clear-text or SSL protocol begins the authentication process by emitting the <request-login> tag element within an <rpc> tag element. In the <request-login> tag element, it encloses the <username> tag element to specify the name of the JUNOS user account under which to establish the connection. The account must already exist on the JUNOScript server machine. You can choose whether or not the application provides the password for the account as part of the initial tag sequence:

- To provide the password along with the JUNOS account name, emit the following tag sequence:

  ```
  <rpc>
    <request-login>
      <username>JUNOS-account</username>
      <challenge-response>password</challenge-response>
    </request-login>
  </rpc>
  ```

  This tag sequence is appropriate if the application automates access to router information and does not interact with users, or obtains the password from a user before beginning the authentication process.

- To omit the password and specify only the JUNOS account name, emit the following tag sequence:

  ```
  <rpc>
      <request-login>
          <username>JUNOS-account</username>
      </request-login>
  </rpc>
  ```

  This tag sequence is appropriate if the application does not obtain the password until the authentication process has already begun. In this case, the JUNOScript server returns the <challenge> tag element within an <rpc-reply> tag element to request the password associated with the account. The tag element encloses the string Password: which the client application can forward to the screen as a prompt for a user. The echo attribute on the <challenge> tag element is set to the value no to specify that the password string typed by the user does not echo on the screen. The tag sequence is as follows:

  ```
  <rpc-reply xmlns:junos="URL">
      <challenge echo="no">Password:</challenge>
  </rpc-reply>
  ```

  The client application obtains the password and emits the following tag sequence to forward it to the JUNOScript server:

  ```
  <rpc>
      <request-login>
          <username>JUNOS-account</username>
          <challenge-response>password</challenge-response>
      </request-login>
  </rpc>
  ```

After it receives the account name and password, the JUNOScript server emits the <authentication-response> tag element to indicate whether the authentication attempt is successful:

- If the password is correct, the authentication attempt succeeds and the JUNOScript server emits the following tag sequence:

  ```
  <rpc-reply xmlns:junos="URL">
      <authentication-response>
          <status>success</status>
          <message>JUNOS-account</message>
      </authentication-response>
  </rpc-reply>
  ```

  The *JUNOS-account* is the JUNOS account name under which the connection is established. The JUNOScript session begins as described in “Start the JUNOScript Session” on page 19.

- If the password is not correct or the <request-login> tag element is otherwise malformed, the authentication attempt fails and the JUNOScript server emits the following tag sequence:

```
<rpc-reply xmlns:junos="URL">
    <authentication-response>
        <status>fail</status>
        <message>error-message</message>
    </authentication-response>
</rpc-reply>
```

The *error-message* is a string explaining why the authentication attempt failed. The JUNOScript server emits the <challenge> tag element up to two more times before rejecting the authentication attempt and closing the connection.

Exchange Tagged Data

The session continues when the client application sends a request to the JUNOScript server. The JUNOScript server does not emit any tag elements after session initialization, except in response to the client application's requests (or in the rare case that it needs to terminate the JUNOScript session). The following sections describe the exchange of tagged data:

- Send a Request to the JUNOScript Server on page 26
- Parse the JUNOScript Server Response on page 28

Send a Request to the JUNOScript Server

To initiate a request to the JUNOScript server, emit the opening <rpc> tag, followed by one or more tag elements that represent the particular request, and the closing </rpc> tag, in that order. Enclose each request in a separate pair of opening <rpc> and closing </rpc> tags. For an example of emitting an <rpc> tag element in the context of a complete JUNOScript session, see "Example of a JUNOScript Session" on page 33.

See the following sections for further information:

- JUNOScript Request Classes on page 27
- Include Attributes in the Opening <rpc> Tag on page 28

JUNOScript Request Classes

There are two classes of JUNOScript requests:

- *Operational requests*—Requests for information about router status, which correspond to the JUNOS CLI commands listed in the JUNOS Internet software operational mode command references. The JUNOScript API defines a specific request tag element for many CLI commands. For example, the <get-interface-information> tag element corresponds to the show interfaces command, and the <get-chassis-inventory> tag element requests the same information as the show chassis hardware command.

  The following sample request is for detailed information about the interface called ge-2/3/0:

  ```
  <rpc>
     <get-interface-information>
        <interface-name>ge-2/3/0</interface-name>
        <detail/>
     </get-interface-information>
  </rpc>
  ```

  For more information about requesting operational information, see "Operational Requests" on page 37. For a complete list of mappings between tag elements and CLI commands for the current version of the JUNOScript API, see the *JUNOScript API Reference*.

- *Configuration requests*—Requests to change router configuration or for information about the current configuration, either candidate or committed (the one currently in active use on the router). The candidate and committed configurations diverge when there are uncommitted changes to the candidate configuration.

  Configuration requests correspond to the JUNOS CLI configuration statements described in each of the JUNOS Internet software configuration guides. The JUNOScript API defines a tag element for every container and leaf statement in the JUNOS configuration hierarchy.

  The following example requests the information about the [edit system login] level of the current candidate configuration:

  ```
  <rpc>
     <get-configuration>
        <configuration>
           <system>
              <login/>
           </system>
        </configuration>
     </get-configuration>
  </rpc>
  ```

  For more information about router configuration, see "Router Configuration" on page 41. For a summary of the available configuration tag elements, see the *JUNOScript API Reference*.

Note

Although operational and configuration requests conceptually belong to separate classes, a JUNOScript session does not have distinct modes that correspond to CLI operational and configuration modes. Each request tag element is enclosed within its own <rpc> tag element, so a client application can freely alternate operational and configuration requests.

The client application can send only one request tag element at a time to a particular router, and must not send another request tag element until it receives the closing </rpc-reply> tag that represents the end of the JUNOScript server response to the current request.

Include Attributes in the Opening <rpc> Tag

Optionally, a client application can include one or more attributes in the opening <rpc> tag for each request. The client application can freely define attribute names, except as described in the following note. The JUNOScript server echoes each attribute, unchanged, in the opening <rpc-reply> tag in which it encloses its response. You can use this feature to associate requests and responses by defining an attribute in each opening request tag that assigns a unique identifier. The JUNOScript server echoes the attribute in its opening <rpc-reply> tag, making it easy to map the response to the initiating request.

Note

The xmlns:junos attribute name is reserved. The JUNOScript server sets the attribute to an appropriate value on the opening <rpc-reply> tag, so client applications must not emit it on the opening <rpc> tag. For more information, see “The xmlns:junos Attribute” on page 29.

Parse the JUNOScript Server Response

The JUNOScript server encloses its response to a client request in an <rpc-reply> tag element. Client applications must include code for parsing the stream of response tags coming from the JUNOScript server, either processing them as they arrive or storing them until the response is complete. See the following sections for further information:

- The xmlns:junos Attribute on page 29
- JUNOScript Server Response Classes on page 29
- Use a Standard API to Parse Response Tag Elements on page 31

The xmlns:junos Attribute

The JUNOScript server includes the xmlns:junos attribute on the opening <rpc-reply> tag to define the XML namespace for the enclosed JUNOScript tag elements that have the junos: prefix on their names. The namespace is a URL of the form http://xml.juniper.net/junos/*release-code*/junos, where *release-code* is the standard string that represents the release of the JUNOS Internet software that is running on the JUNOScript server machine.

In the following example, the namespace corresponds to the initial version of JUNOS Release 5.6:

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/5.6R1/junos">
```

JUNOScript Server Response Classes

There are three classes of JUNOScript server responses:

- *Operational responses*—Responses to requests for information about router status. These correspond to the output from JUNOS CLI operational commands as described in the JUNOS Internet software operational mode command references. The JUNOScript API defines response tag elements for all defined JUNOScript operational request tag elements.

  For example, the JUNOScript server returns the information requested by the <get-interface-information> tag element in a response tag element called <interface-information>, and the information requested by the <get-chassis-inventory> tag element in a response tag element called <chassis-inventory>.

  The opening tag for an operational response usually includes the xmlns attribute to define the XML namespace for the enclosed tag elements that do not have a prefix (such as junos:) before their names. The namespace is a URL of the form http://xml.juniper.net/junos/*release-code*/junos-*category*, where *release-code* is the standard string that represents the release of the JUNOS Internet software that is running on the JUNOScript server machine, and *category* represents the type of information.

  The following sample response includes information about the interface called ge-2/3/0. The namespace indicated by the xmlns attribute contains interface information for the initial version of JUNOS Release 5.6. (Note that the opening <interface-information> tag element appears on two lines only for legibility.)

  ```
  <rpc-reply xmlns:junos="URL">
    <interface-information xmlns="http://xml.juniper.net/junos/5.6R1/junos-interface">
      <physical-interface>
        <name>ge-2/3/0</name>
        <!-- other data tag elements for the ge-2/3/0 interface -->
      </physical-interface>
    </interface-information>
  </rpc-reply>
  ```

  For more information about the contents of operational response tag elements, see “Operational Requests” on page 37. For a summary of operational response tag elements, see the *JUNOScript API Reference*.

- *Configuration information responses*—Responses to requests for information about the router's current configuration. The JUNOScript API defines a tag element for every container and leaf statement in the JUNOS configuration hierarchy.

  The following sample response includes the information at the [edit system login] level of the configuration hierarchy. For brevity, the sample shows only one user defined at this level.

  ```
  <rpc-reply xmlns:junos="URL">
      <configuration>
          <system>
              <login>
                  <user>
                      <name>admin</name>
                      <full-name>Administrator</full-name>
                      <!-- other data tag elements for the admin user -->
                  </user>
              </login>
          </system>
      </configuration>
  </rpc-reply>
  ```

  For more information about router configuration, see "Router Configuration" on page 41. For a summary of the available configuration tag elements, see the *JUNOScript API Reference.*

- *Configuration change responses*—Responses to requests to change router configuration. For commit operations, the JUNOScript server encloses an explicit indicator of success or failure within the <commit-results> tag element. For other operations, the JUNOScript server indicates success by returning an opening <rpc-reply> and closing </rpc-reply> tag with nothing between them, rather than emitting an explicit success indicator.

  For more information about router configuration, see "Router Configuration" on page 41. For a summary of the available configuration tag elements, see the *JUNOScript API Reference.*

For an example of parsing the <rpc-reply> tag element in the context of a complete JUNOScript session, see "Example of a JUNOScript Session" on page 33.

Use a Standard API to Parse Response Tag Elements

Client applications can handle incoming XML tag elements by feeding them to a parser that implements a standard API such as the Document Object Model (DOM) or Simple API for XML (SAX).

Routines in the DOM accept incoming XML and build a tag hierarchy in the client application's memory. There are also DOM routines for manipulating an existing hierarchy. DOM implementations are available for several programming languages, including C, C++, Perl, and Java. The DOM specification is available at http://www.w3.org/TR/REC-DOM-Level-1. Additional information is available at http://search.cpan.org/author/TJMATHER/XML-DOM-1.41/lib/XML/DOM.pm (part of the Comprehensive Perl Archive Network [CPAN] Web site).

Note The version indicator in the URL for the DOM.pm file (1.41 in the preceding URL) is subject to frequent revision. Try substituting higher version numbers if the file cannot be found.

One potential drawback with DOM is that it always builds a hierarchy of tag elements, which can become very large. If a client application needs to handle only a subhierarchy at a time, it can use a parser that implements SAX instead. SAX accepts XML and feeds the tag elements directly to the client application, which must build its own tag hierarchy. For more information about SAX, see http://sax.sourceforge.net.

End the Session and Close the Connection

When a client application is finished making requests, it ends the JUNOScript session by emitting the empty <request-end-session/> tag within an <rpc> tag element. In response, the JUNOScript server emits the <end-session/> tag enclosed in an <rpc-reply> tag element. The client application waits to receive this reply before emitting its closing </junoscript> tag. For an example of the exchange of closing tags, see "Example of a JUNOScript Session" on page 33.

The client application can then close the ssh, SSL, or other connection to the JUNOScript server machine. Client applications written in Perl can close the JUNOScript session and connection by using the JUNOScript Perl module described in "Write Perl Client Applications" on page 81. For more information, see that chapter.

Client applications that do not use the JUNOScript Perl module use the routine provided for closing a connection in the standard library for their programming language.

Handle an Error Condition

If the JUNOScript server encounters an error condition that prevents it from processing the current request, it emits an <xnm:error> tag element, which encloses child tag elements that describe the nature of the error. Client applications must be prepared to receive and handle an <xnm:error> tag element at any time. The information in any response tag elements already received that are related to the current request might be incomplete. The client application can include logic for deciding whether to discard or retain the information.

An error can occur while the server is performing any of the following operations, and the server can send a different combination of child tag elements in each case:

- Processing a request submitted by a client application in a defined request tag element
- Processing a command string submitted by a client application in a <command> tag element (discussed in "Requests and Responses without Defined JUNOScript Tag Elements" on page 39)
- Opening, locking, committing, or closing a configuration as requested by a client application (discussed in "Router Configuration" on page 41)
- Parsing a configuration file submitted by a client application in a <load-configuration> tag element (discussed in "Change the Candidate Configuration" on page 61)

If the JUNOScript server encounters a less serious problem, it can emit an <xnm:warning> tag element instead. The usual response for the client application in this case is to log the warning or pass it to the user, but to continue parsing the server's response.

For a description of the child tag elements that can appear within an <xnm:error> or <xnm:warning> tag element to specify the nature of the problem, see the entries for <xnm:error> and <xnm:warning> in the *JUNOScript API Reference*.

Halt a Request

To request that the JUNOScript server stop processing the current request, emit the empty <abort/> tag. The JUNOScript server responds with the empty <abort-acknowledgment/> tag. Depending on the operation being performed, response tag elements already sent by the JUNOScript server for the halted request are possibly invalid. The client application can include logic for deciding whether to discard or retain them as appropriate.

For more information about the <abort/> and <abort-acknowledgment/> tags, see their entries in the *JUNOScript API Reference*.

Display CLI Output as JUNOScript Tag Elements

To display the output from a JUNOS CLI command as JUNOScript tag elements rather than the default formatted ASCII, pipe the command to the display xml command. The following example shows the output from the show chassis hardware command issued on an M40 Internet router that is running the initial version of JUNOS Release 5.6:

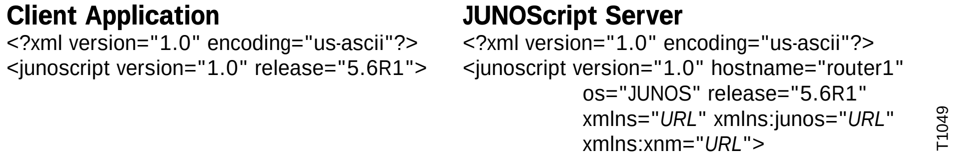
```
user@host> show chassis hardware | display xml
<rpc-reply xmlns:junos=”http://xml.juniper.net/junos/5.6R1/junos”>
<chassis-inventory xmlns="http://xml.juniper.net/junos/5.6R1/junos-chassis">
<chassis junos:style="inventory">
<name>Chassis</name>
<serial-number>00118</serial-number>
<description>M40</description>
<chassis-module>
<name>Backplane</name>
<version>REV 06</version>
<part-number>710-000073</part-number>
<serial-number>AA2049</serial-number>
</chassis-module>
<chassis-module>
<name>Power Supply A</name>
...
</chassis-module>
...
</chassis>
</chassis-inventory>
</rpc-reply>
```

Example of a JUNOScript Session

This section describes the sequence of tag elements in a sample JUNOScript session. The client application begins by establishing a connection to a JUNOScript server.

Exchange Initialization PIs and Tag Elements

The two parties then exchange initialization PIs and tag elements, as shown in the following example. Note that the JUNOScript server’s opening <junoscript> tag appears on multiple lines for legibility only. The server does not actually insert a newline character into the list of attributes. For a detailed discussion of the <?xml?> PI and opening <junoscript> tag, see “Start the JUNOScript Session” on page 19.


Client Application

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" release="5.6R1">
```

JUNOScript Server

```
<?xml version="1.0" encoding="us-ascii"?>
<junoscript version="1.0" hostname="router1"
            os="JUNOS" release="5.6R1"
            xmlns="URL" xmlns:junos="URL"
            xmlns:xnm="URL">
```

Send an Operational Request

The client application now emits the <get-chassis-inventory> tag element to request information about the router's chassis hardware. The JUNOScript server returns the requested information in the <chassis-inventory> tag element. In the following example, tag elements appear indented and on separate lines for legibility only. Client applications do not need to include newlines, spaces, or other white space characters in the tag stream they send to the JUNOScript server, because the server automatically discards such characters. Also, client applications can issue all tag elements that constitute a request within a single routine such as the C-language write( ) routine, or can invoke a separate routine for each tag element or group of tag elements.

```
Client Application          JUNOScript Server
<rpc>
  <get-chassis-inventory>
    <detail/>
  </get-chassis-inventory>
</rpc>
                            <rpc-reply xmlns:junos="URL">
                              <chassis-inventory xmlns="URL">
                                <chassis>
                                  <name>Chassis</name>
                                  <serial-number>1122</serial-number>
                                  <description>M10</description>
                                  <chassis-module>
                                    <name>Midplane</name>
                                    <!- - other child tags for the Midplane chassis module - ->
                                  </chassis-module>
                                  <!- - tags for other chassis modules - ->
                                </chassis>
                              </chassis-inventory>
                            </rpc-reply>
                                                                              T1002
```

Lock the Configuration

The client application then prepares to create a new privilege class called network-mgmt at the [edit system login class] level of the configuration hierarchy. It begins by using the <lock-configuration/> tag to prevent any other users or applications from altering the candidate configuration at the same time. To confirm that the candidate configuration is locked, the JUNOScript server returns an <rpc-reply> and an </rpc-reply> tag with nothing between them.

```
Client Application        JUNOScript Server
<rpc>
  <lock-configuration/>
</rpc>
                          <rpc-reply xmlns:junos="URL"></rpc-reply>
                                                                              T1003
```

Change the Configuration

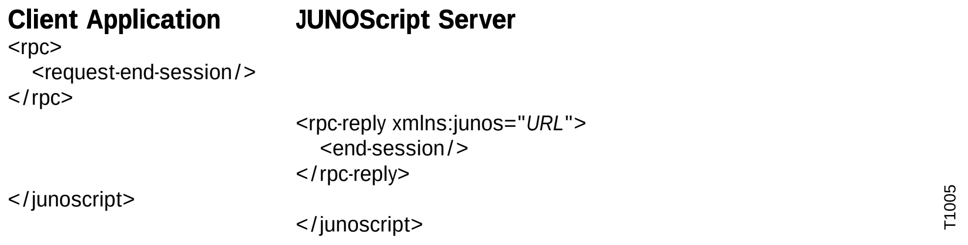
The client application emits the tag elements that define the new network-mgmt privilege class, commits them, and unlocks (and by implication closes) the configuration. As when it opens the configuration, the JUNOScript server confirms successful receipt and closure of the configuration only by returning an opening <rpc-reply> and closing </rpc-reply> tag with nothing between them, not with a more explicit signal. It returns the <commit-results> tag element to report the outcome of the commit operation. (You do not need to understand the meaning of all tag elements at this point. For more information about configuring a router, see “Router Configuration” on page 41.)

```
Client Application                          JUNOScript Server
<rpc>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <class>
            <name>network-mgmt</name>
            <permissions>configure</permissions>
            <permissions>snmp</permissions>
            <permissions>system</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
                                            <rpc-reply xmlns:junos="URL">
                                              <load-configuration-results>
                                                <load-success/>
                                              </load-configuration-results>
                                            </rpc-reply>
<rpc>
  <commit-configuration/>
</rpc>
                                            <rpc-reply xmlns:junos="URL">
                                              <commit-results>
                                                <routing-engine>
                                                  <name>re0</name>
                                                  <commit-success/>
                                                </routing-engine>
                                              </commit-results>
                                            </rpc-reply>
<rpc>
  <unlock-configuration/>
</rpc>
                                            <rpc-reply xmlns:junos="URL"></rpc-reply>
```

T1042

Close the JUNOScript Session

The client application closes the JUNOScript session:



Client Application **JUNOScript Server**

```
<rpc>
    <request-end-session/>
</rpc>
                               <rpc-reply xmlns:junos="URL">
                                   <end-session/>
                               </rpc-reply>
</junoscript>
                               </junoscript>
```

Chapter 3
Operational Requests

This chapter explains how to use the JUNOScript application programming interface (API) to request information about router status. The JUNOScript request tag elements described here correspond to JUNOS command-line interface (CLI) operational commands described in the JUNOS Internet software operational mode command references. The JUNOScript API defines a specific request tag element for most commands in the CLI show family of commands. For example, the <get-interface-information> tag element corresponds to the show interfaces command, and the <get-chassis-inventory> tag element requests the same information as the show chassis hardware command.

This chapter discusses the following topics:

- Request Operational Information on page 37
- Parse an Operational Response on page 39
- Requests and Responses without Defined JUNOScript Tag Elements on page 39

For information about using the JUNOScript API to request router configuration information, see “Router Configuration” on page 41.

Request Operational Information

To request operational information from the JUNOScript server, first establish a connection to it and open a JUNOScript session, as described in “Connect to the JUNOScript Server” on page 18 and “Start the JUNOScript Session” on page 19.

Then emit operational request tag elements, each enclosed in an opening <rpc> and closing </rpc> tag. Client applications can emit an unlimited number of operational request tag elements during a JUNOScript session (one at a time). It can freely intermingle them with configuration requests, which are described in “Router Configuration” on page 41.

Each operational request tag element corresponds to a JUNOS CLI command that has a distinct function or returns a different kind of output. For a list of mappings between CLI commands and operational request tag elements, see the *JUNOScript API Reference*.

The JUNOScript API defines separate Extensible Markup Language (XML) document type definition (DTD) files for different JUNOS components. For instance, the DTD for interface information is called junos-interface.dtd and the DTD for chassis information is called junos-chassis.dtd. Each DTD constitutes a separate XML namespace, which means that multiple DTDs can define a tag element with the same name but a distinct function. The *JUNOScript API Reference* includes the text of the JUNOScript DTDs.

Client applications can emit all tag elements that constitute a request by invoking one instance of a routine such as write( ), or can invoke a separate instance of the routine for each tag element or group of tag elements. The JUNOScript server ignores any newline characters, spaces, or other white space characters in the tag stream.

After the client application finishes making requests, it can close the JUNOScript session and terminate the connection, as described in "End the Session and Close the Connection" on page 31.

JUNOS CLI commands take two main kinds of options, and there are matching child tag elements in the corresponding operational request tag element. For a discussion and example of each, see the following sections:

- Map Child Tag Elements to Options with Variable Values on page 38
- Map Child Tag Elements to Fixed-Form Options on page 39

Map Child Tag Elements to Options with Variable Values

Many JUNOS CLI commands have options that specify the name of the object that the command affects or reports about. In some cases, the CLI does not precede the option name with a fixed-form keyword, but XML convention requires that the JUNOScript API define a tag element for each option. The tag element is most often called <name>, but other names are sometimes used. To learn the names for the child tag elements of an operational request tag element, consult the tag element's entry in the appropriate DTD or in the *JUNOScript API Reference.*

The following illustrates the mapping between CLI command and JUNOScript tag elements for two CLI operational commands with variable-form options. In the show interfaces command, t3-5/1/0:0 is the name of the interface. In the show bgp neighbor command, 10.168.1.222 is the IP address for the BGP peer of interest.

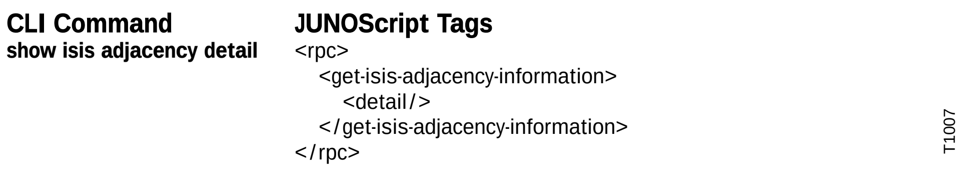
```
CLI Command                      JUNOScript Tags
show interfaces t3-5/1/0:0       <rpc>
                                   <get-interface-information>
                                     <interface-name>t3-5:1/0:0</interface-name>
                                   </get-interface-information>
                                 </rpc>
show bgp neighbor 10.168.1.122   <rpc>
                                   <get-bgp-neighbor-information>
                                     <neighbor-address>10.168.1.122</neighbor-address>
                                   </get-bgp-neighbor-information>
                                 </rpc>
```

T1006

Map Child Tag Elements to Fixed-Form Options

Some JUNOS CLI commands include options that have a fixed form, such as the brief and detail strings, which specify the amount of detail to include in the output. The JUNOScript API usually maps such an option to an empty tag element whose name matches the option name.

The following illustrates the mapping between the CLI command and JUNOScript tag elements for the show isis adjacency command, which has a fixed-form option called detail:



CLI Command
show isis adjacency detail

JUNOScript Tags

```
<rpc>
   <get-isis-adjacency-information>
      <detail/>
   </get-isis-adjacency-information>
</rpc>
```

T1007

Parse an Operational Response

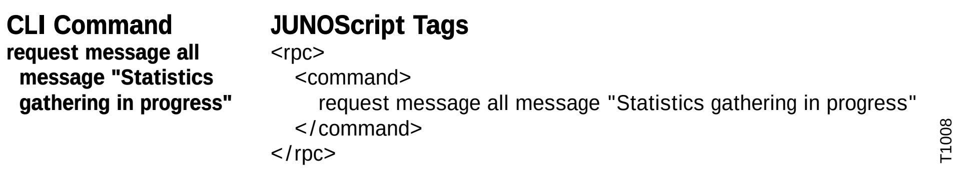
The JUNOScript server encloses its response to each operational request in a separate <rpc-reply> tag element. For information about the conventions that client applications must follow when interpreting a JUNOScript server response, see “General JUNOScript Conventions” on page 9. For information about parsing a response tag element, see “Parse the JUNOScript Server Response” on page 28.

Client applications must include code for accepting the tag stream and extracting the content from tag elements. They can, for instance, feed them to a parser that implements a standard API such as the Document Object Model (DOM) or Simple API for XML (SAX). For more information, see “Use a Standard API to Parse Response Tag Elements” on page 31.

Requests and Responses without Defined JUNOScript Tag Elements

The JUNOScript API does not define a tag element for every JUNOS CLI command. Client applications can still invoke the functionality of such commands by enclosing the actual CLI command string in a <command> tag element. Like all request tag elements, the <command> tag element must occur within an <rpc> tag element.

The following illustrates the use of the <command> tag element to issue the request message command, for which the JUNOScript API does not define a tag element. The line breaks in both the CLI command and within the <command> tag elements are for legibility only. The client application should not include newlines or other extraneous white space characters in the string within the <command> tag elements. Also, unlike most examples in this manual, the following does not preserve a line-to-line correspondence between the CLI command and the JUNOScript tag elements:



CLI Command
request message all
message "Statistics
gathering in progress"

JUNOScript Tags

```
<rpc>
   <command>
      request message all message "Statistics gathering in progress"
   </command>
</rpc>
```

T1008

If the JUNOScript API does not define a response tag element for the type of output requested by a client application, the JUNOScript server encloses its response in an <output> tag element. The tag element's contents are usually one or more lines of formatted ASCII output like that displayed by the JUNOS CLI on the computer screen.

Caution

The JUNOScript server might not support use of all CLI commands in the <command> tag element. For details, see the JUNOS Internet software release notes.

The content and formatting of data within an <output> tag element are subject to change, so client applications must not depend on them. Future versions of the JUNOScript API will define specific response tag elements (rather than <output> tag elements) for more commands. Client applications that rely on the content of <output> tag elements will not be able to interpret the output from future versions of the JUNOScript server.

Chapter 4 Router Configuration

This chapter explains how to use the JUNOScript application programming interface (API) to change router configuration or to request information about the current configuration. The JUNOScript tag elements described here correspond to JUNOS command-line interface (CLI) configuration statements described in the JUNOS Internet software configuration guides.

This chapter discusses the following topics:

Mapping between CLI Configuration Statements and JUNOScript Tag Elements

The JUNOScript API defines a tag element for every container and leaf statement in the JUNOS configuration hierarchy. At the top levels of the configuration hierarchy, there is almost always a one-to-one mapping between tag elements and statements, and most tag names match the configuration statement name. At deeper levels of the configuration hierarchy, the mapping is sometimes less direct, because some CLI notational conventions do not map directly to Extensible Markup Language (XML)-compliant tagging syntax. The following sections describe the mapping between configuration statements and JUNOScript tag elements:

- Tag Element Mappings for Top-Level (Container) Statements on page 42
- Tag Element Mappings for Leaf Statements on page 43
- Tag Element Mappings for Identifiers on page 44
- Tag Element Mappings for Leaf Statements with Multiple Values on page 45
- Tag Element Mappings for Multiple Options on One or More Lines on page 46
- Tag Element Mapping for Comments about Configuration Statements on page 47

Note

For some configuration statements, the notation used when typing the statement at the CLI configuration-mode prompt differs from the notation used in a configuration file. The same JUNOScript tag element maps to both notational styles.

Tag Element Mappings for Top-Level (Container) Statements

The <configuration> tag element is the top-level JUNOScript container tag element for configuration statements, and corresponds to the CLI [edit] level. Most statements at the next few levels of the configuration hierarchy are container statements, and the name of each corresponding JUNOScript container tag element almost always matches the container statement name.

Note

The top-level <configuration-text> tag element also corresponds to the CLI configuration mode's [edit] level. It encloses formatted ASCII configuration statements instead of JUNOScript tag elements, and is not relevant to the following discussion. For more information, see "Specify Formatted ASCII or JUNOScript-Tagged Output" on page 52.

The following depicts the mappings for two sample statements that begin at the top level of the configuration hierarchy. Note how a closing brace in a CLI configuration statement corresponds to a closing JUNOScript tag.

```
CLI Configuration Statements        JUNOScript Tags
                                    <configuration>
system {                              <system>
  login {                               <login>
     …child statements…                   <!- - child statements - ->
  }                                     </login>
}                                     </system>
protocols {                           <protocols>
  ospf {                                <ospf>
     …child statements…                   <!- - child statements - ->
  }                                     </ospf>
}                                     </protocols>
                                    </configuration>
```

T1009

Tag Element Mappings for Leaf Statements

A *leaf statement* is a CLI configuration statement that does not contain any other statements. Most leaf statements define a value for one characteristic of a configuration object and have the following form:

keyword *value*;

In general, the name of the JUNOScript tag element corresponding to a leaf statement is the same as the keyword string. The string between the opening and closing JUNOScript tags is the *value*.

The following depicts the mappings for two sample leaf statements that have a keyword and a value: the message statement at the [edit system login] level and the preference statement at the [edit protocols ospf] level:

```
CLI Configuration Statements           JUNOScript Tags
                                       <configuration>
system {                                 <system>
  login {                                  <login>
     message "Authorized users only";        <message>Authorized users only</message>
     …other statements under login …         <!- - other children of the <login> tag - ->
  }                                        </login>
}                                        </system>
protocols {                              <protocols>
  ospf {                                   <ospf>
     preference 15;                          <preference>15</preference>
     …other statements under ospf …          <!- - other children of the <ospf> tag - ->
  }                                        </ospf>
}                                        </protocols>
                                       </configuration>
```

T1010

Some leaf statements consist of a fixed-form keyword only, without an associated variable-form value. The JUNOScript API represents such statements with an empty tag. The following example shows the mapping for the disable statement at the [edit forwarding-options sampling] hierarchy level:

CLI Configuration Statement

```
forwarding-options {
    sampling {
        disable;
        …other statements under sampling …
    }
}
```

JUNOScript Tags

```
<configuration>
    <forwarding-options>
        <sampling>
            <disable/>
            <!- - other children of the <sampling> tag - ->
        </sampling>
    </forwarding-options>
</configuration>
```

T1011

Tag Element Mappings for Identifiers

At some hierarchy levels, the same kind of container object can appear multiple times. Each instance of the object has a unique identifier to distinguish it from the other instances. In the JUNOS CLI notation, the first statement in the set of statements for such an object consists of a keyword and identifier of the following form:

```
keyword identifier {
    … configuration statements for individual characteristics …
}
```

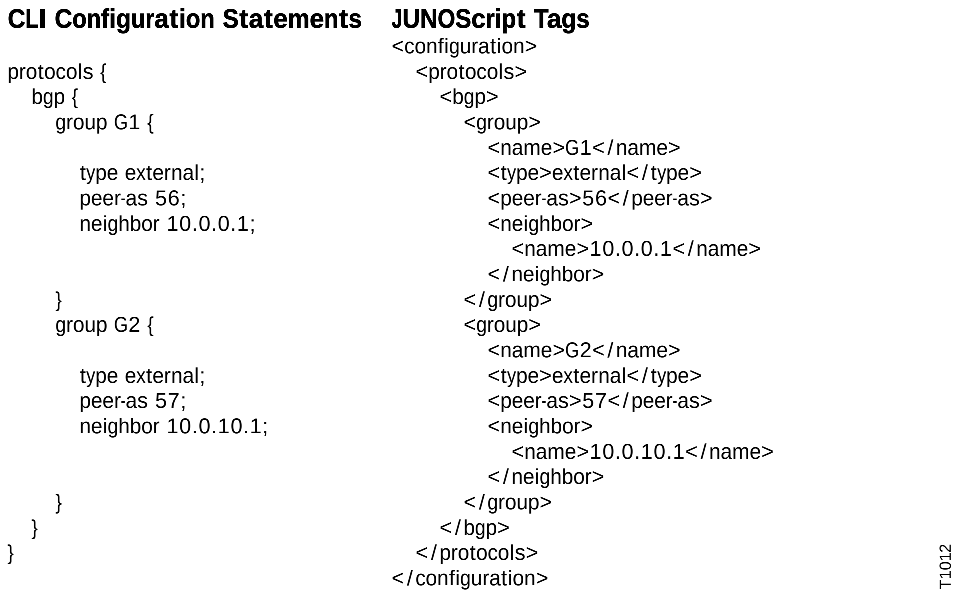
The keyword is a fixed string that indicates the type of object being defined, and the *identifier* is the unique name for this instance of the type. In the JUNOScript API, the tag element corresponding to the keyword is a container tag element for child tag elements that represent the object's characteristics. The container tag element's name generally matches the keyword string.

The JUNOScript API differs from the CLI in its treatment of the identifier. Because the JUNOScript API does not allow container tag elements to contain both other tag elements and untagged character data such as an identifier name, the identifier must be enclosed in a tag element of its own. Most frequently, identifier tag elements are called <name>. To verify the identifier tag element name for an object, consult the object's entry in the appropriate document type definition (DTD) or in the *JUNOScript API Reference*.

Identifier tag elements also constitute an exception to the general XML convention that tag elements at the same level of hierarchy can appear in any order; the identifier tag element always occurs first within the container tag element.

The [edit protocols bgp group] statement is an example of a configuration statement with an identifier. Each Border Gateway Protocol (BGP) group has an associated name (the identifier) and can have other characteristics such as a type, peer autonomous system (AS) number, and neighbor address.

The following example illustrates the mapping between the CLI statements and JUNOScript tag elements that create two BGP groups called G1 and G2. Notice how the JUNOScript <name> tag element that encloses the identifier for each group (and for the identifier of the neighbor within a group) does not have a counterpart in the CLI statements. For complete information about changing router configuration, see “Change the Candidate Configuration” on page 61.



```
CLI Configuration Statements        JUNOScript Tags
                                    <configuration>
protocols {                           <protocols>
  bgp {                                 <bgp>
    group G1 {                            <group>
                                            <name>G1</name>
      type external;                        <type>external</type>
      peer-as 56;                           <peer-as>56</peer-as>
      neighbor 10.0.0.1;                    <neighbor>
                                              <name>10.0.0.1</name>
                                            </neighbor>
    }                                     </group>
    group G2 {                            <group>
                                            <name>G2</name>
      type external;                        <type>external</type>
      peer-as 57;                           <peer-as>57</peer-as>
      neighbor 10.0.10.1;                   <neighbor>
                                              <name>10.0.10.1</name>
                                            </neighbor>
    }                                     </group>
  }                                     </bgp>
}                                     </protocols>
                                    </configuration>
```

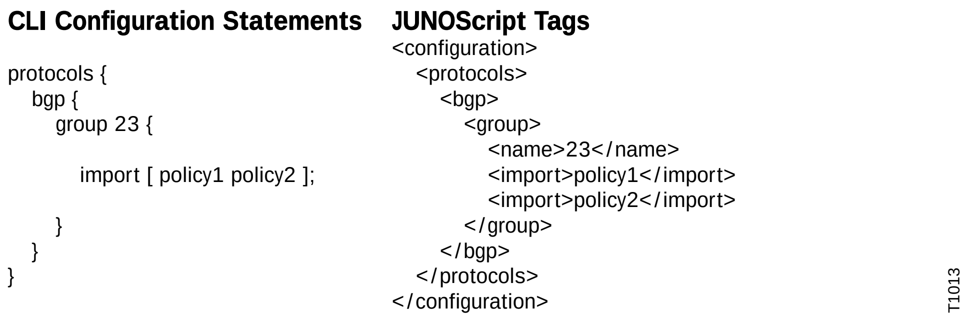
T1012

Tag Element Mappings for Leaf Statements with Multiple Values

Some JUNOS CLI leaf statements accept multiple values, which can either be user-defined or drawn from a set of predefined values. CLI notation uses square brackets to enclose all values in a single statement, as in the following:

statement [*value1 value2 value3*];

The JUNOScript API instead encloses each value in its own tag element. The following example illustrates the mapping between a CLI statement with multiple user-defined values and the corresponding JUNOScript tag elements. The import statement imports two routing policies defined elsewhere in the configuration. For complete information about changing router configuration, see “Change the Candidate Configuration” on page 61.



```
CLI Configuration Statements        JUNOScript Tags
                                    <configuration>
protocols {                           <protocols>
  bgp {                                 <bgp>
    group 23 {                            <group>
                                            <name>23</name>
      import [ policy1 policy2 ];           <import>policy1</import>
                                            <import>policy2</import>
    }                                     </group>
  }                                     </bgp>
}                                     </protocols>
                                    </configuration>
```

T1013

The following example illustrates the same mapping for a CLI statement with multiple predefined values. The permissions statement grants three predefined JUNOS permissions to members of the user-accounts login class.

CLI Configuration Statements

```
system {
    login {
        class user-accounts {

            permissions [ configure admin control ];



        }
    }
}
```

JUNOScript Tags

```
<configuration>
    <system>
        <login>
            <class>
                <name>user-accounts</name>
                <permissions>configure</permissions>
                <permissions>admin</permissions>
                <permissions>control</permissions>
            </class>
        </login>
    </system>
</configuration>
```

T1014

Tag Element Mappings for Multiple Options on One or More Lines

For some configuration objects, the configuration file specifies the value for more than one option on a single line, usually for greater legibility and conciseness. In most such cases, the first option identifies the object and does not have a keyword, but later options are paired keywords and values. The JUNOScript API encloses each option in its own tag element. Because the first option has no keyword in the CLI statement, the JUNOScript API assigns a name to its tag element.

The following example illustrates the mapping of a CLI configuration statement that places multiple options on a single line to the corresponding JUNOScript tag elements. Notice that the JUNOScript API defines a tag element for both options and assigns a name to the tag element for the first option (10.0.0.1), which has no CLI keyword.

CLI Configuration Statements

```
system {
    backup-router 10.0.01 destination 10.0.0.2;



}
```

JUNOScript Tags

```
<configuration>
    <system>
        <backup-router>
            <address>10.0.0.1</address>
            <destination>10.0.0.2</destination>
        </backup-router>
    </system>
</configuration>
```

T1015

The configuration file entries for some configuration objects have multiple lines with more than one option, and again the JUNOScript API defines a separate tag element for each option. The following example illustrates the mappings for a sample [edit protocols isis traceoptions] statement, which contains three statements, each with multiple options:

```
CLI Configuration Statements                        JUNOScript Tags
                                                    <configuration>
protocols {                                           <protocols>
  isis {                                                <isis>
    traceoptions {                                        <traceoptions>
      file trace-file size 3m files 10 world-readable;      <file>
                                                              <filename>trace-file</filename>
                                                              <size>3m</size>
                                                              <files>10</files>
                                                              <world-readable/>
                                                            </file>
      flag route detail;                                    <flag>
                                                              <name>route</name>
                                                              <detail/>
                                                            </flag>
      flag state receive;                                   <flag>
                                                              <name>state</name>
                                                              <receive/>
                                                            </flag>
    }                                                     </traceoptions>
  }                                                     </isis>
}                                                     </protocols>
                                                    </configuration>
```

T1016

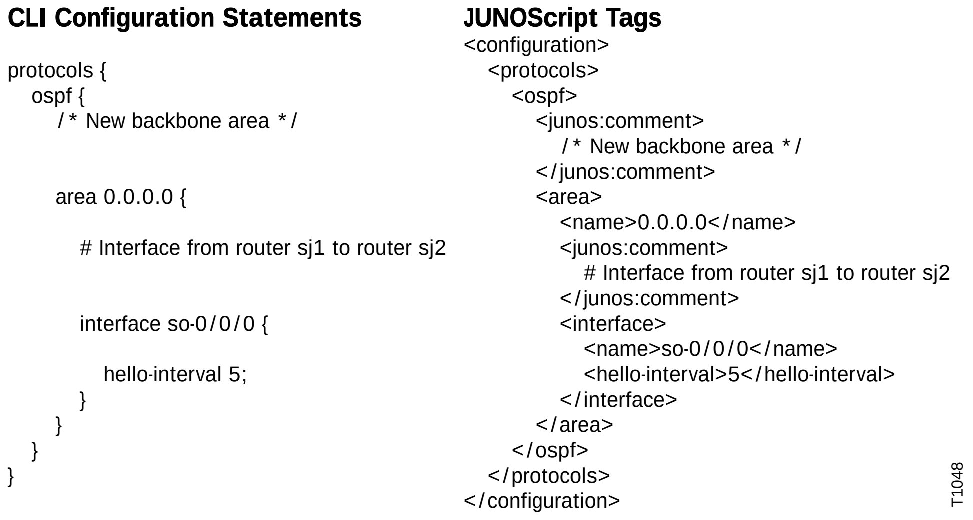
Tag Element Mapping for Comments about Configuration Statements

The JUNOS configuration database can include comments that describe statements in the configuration. In CLI configuration mode, the annotate command specifies the comment to associate with a statement at the current hierarchy level. Comments can also be inserted directly into the ASCII version of the configuration file using a text editor. For more information, see the *JUNOS Internet Software Configuration Guide: Getting Started*.

The JUNOScript API encloses comments about configuration statements in the <junos:comment> tag element. (These comments are different than those described in "XML Comments" on page 12, which are enclosed in the strings <!-- and --> and are automatically discarded by the JUNOScript server.)

Place the <junos:comment> tag element immediately before the tag element that represents the configuration statement to associate with the comment. (If the tag element for the associated statement is omitted, the comment is not recorded in the configuration database.) The comment text string should not include any linebreak characters, but should include one of the two delimiters that indicate a comment in the configuration database: either the # character before the comment or the paired strings /* before the comment and */ after it.

The following example illustrates how to associate comments with the mappings for two statements in a sample [edit protocols ospf area interface] hierarchy:



```
CLI Configuration Statements               JUNOScript Tags
                                           <configuration>
protocols {                                  <protocols>
  ospf {                                       <ospf>
    /* New backbone area */                      <junos:comment>
                                                   /* New backbone area */
                                                 </junos:comment>
    area 0.0.0.0 {                               <area>
                                                   <name>0.0.0.0</name>
      # Interface from router sj1 to router sj2    <junos:comment>
                                                     # Interface from router sj1 to router sj2
                                                   </junos:comment>
      interface so-0/0/0 {                         <interface>
                                                     <name>so-0/0/0</name>
        hello-interval 5;                            <hello-interval>5</hello-interval>
      }                                            </interface>
    }                                            </area>
  }                                            </ospf>
}                                            </protocols>
                                           </configuration>
```

T1048

Same Tag Elements Used for Requests and Responses

The JUNOScript server encloses its response to a configuration request in an <rpc-reply> tag element, just as for an operational request. Within the <rpc-reply> tag element, it by default returns a JUNOScript-tagged response enclosed in a <configuration> tag element. (If the client application requests a formatted ASCII response, the server instead encloses the response in a <configuration-text> tag element. For more information, see “Specify Formatted ASCII or JUNOScript-Tagged Output” on page 52.)

Enclosing every JUNOScript-tagged configuration response within a <configuration> tag element contrasts with how the server encloses each different kind of operational response in a tag element named for that type of response—for example, the <chassis-inventory> tag element for chassis information or the <interface-information> tag element for interface information.

The JUNOScript tag elements within the <configuration> tag element represent configuration hierarchy levels, configuration objects, and object characteristics, always ordered from higher to deeper levels of the hierarchy. When a client application loads a configuration, it emits the same tag elements in the same order as the JUNOScript server uses when returning configuration information. This consistent representation helps make handling configuration information more straightforward. For instance, the client application can change router configuration by requesting the current configuration, storing the JUNOScript server’s response to a local memory buffer, making changes or applying transformations to the buffered data, and reloading the altered configuration. Because the altered configuration is based on the JUNOScript server’s response, it is certain to be syntactically correct. For more information about changing router configuration, see “Change the Candidate Configuration” on page 61.

Similarly, when a client application requests information about a configuration hierarchy level or object, it uses the same tag elements that the JUNOScript server will return in response. To represent the hierarchy level or object about which it is requesting information, the client application sends a complete stream of tag elements from the top of the configuration hierarchy (represented by the <configuration> tag element) down to the requested level or object. The innermost tag element, which represents the level or object, is either empty or includes the identifier tag element only. The JUNOScript server's response includes the same stream of tag elements, but the tag element for the requested level or object contains all the tag elements that represent the level's child configuration elements or object's characteristics. For more information about requesting configuration information, see "Request Configuration Information" on page 51.

One potential difference between the tag stream in the JUNOScript server's response and one emitted by a client application concerns the use of white space. By XML convention, the JUNOScript server ignores white space in the tag stream it receives. In the stream that it emits, however, the JUNOScript server includes newline characters and extra spaces between tag elements. If a client application writes the response to a file, each tag element appears on its own line, and child tag elements are indented from their parents, both of which enhance readability for users. Client applications can ignore or discard the white space, particularly if they do not write the tag elements to a file for later review. Client applications do not need to include the white space in requests to the JUNOScript server.

Overview of Router Configuration Procedures

To read or change router configuration information, perform the following procedures, which are described in the indicated sections:

1. Establish a connection to the JUNOScript server on the router, as described in "Connect to the JUNOScript Server" on page 18.

2. Open a JUNOScript session, as described in "Start the JUNOScript Session" on page 19.

3. (Optional) Lock the current candidate configuration, as described in "Lock the Candidate Configuration" on page 50. Locking the configuration prevents other users or applications from changing it at the same time.

4. Request or change configuration information, as described in "Request Configuration Information" on page 51 and "Change the Candidate Configuration" on page 61.

5. (Optional) Verify the syntactic correctness of the candidate configuration before attempting to commit it, as described in "Verify the Syntactic Correctness of the Candidate Configuration" on page 72.

6. Commit changes made to the candidate configuration (if appropriate), as described in "Commit the Candidate Configuration" on page 72.

7. Unlock the current configuration if it is locked, as described in "Unlock the Candidate Configuration" on page 77.

8. End the JUNOScript session and close the connection to the router, as described in "End the Session and Close the Connection" on page 31.

Lock the Candidate Configuration

Before reading or changing router configuration, a client application must establish a connection to the JUNOScript server, as described in “Prerequisites for telnet Connections” on page 17, and open a JUNOScript session, as described in “Start the JUNOScript Session” on page 19.

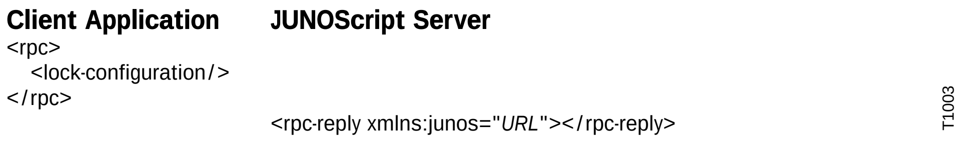
The application can then emit the <lock-configuration/> tag enclosed in an <rpc> tag element if it needs to prevent other users or applications from changing the candidate configuration. If it is not important to block changes from other applications, the application can begin requesting or changing configuration information immediately, as described in “Request Configuration Information” on page 51 and “Change the Candidate Configuration” on page 61.

Only one application can hold the lock on the candidate configuration at a time. Other users and applications can still read the configuration while it is locked. The lock persists until either the JUNOScript session ends or the client application emits the <unlock-configuration/> tag, as described in “Unlock the Candidate Configuration” on page 77.

If the JUNOScript server successfully locks the configuration, it returns an opening <rpc-reply> and closing </rpc-reply> tag with nothing between them. If it cannot lock the configuration, it returns an <xnm:error> tag element instead. Reasons for the failure include the following:

- Another user or application has already locked the candidate configuration. The error message reports the login identity of the user or application.
- The candidate configuration includes changes that have not yet been committed. To commit the changes, see “Commit the Candidate Configuration” on page 72. To roll back to a previous version of the configuration (and lose the uncommitted changes), see “Roll Back to a Previous Configuration” on page 71.

In the following example, the client application emits the <lock-configuration/> tag after opening the JUNOScript session and exchanging initialization information with the JUNOScript server:



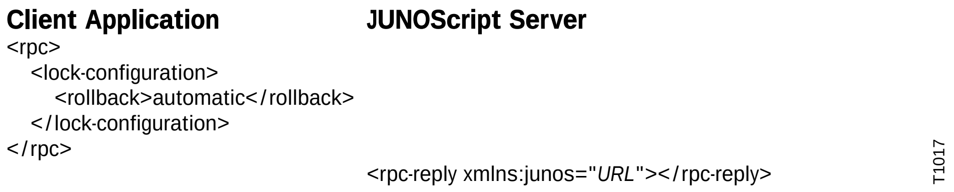
```
Client Application     JUNOScript Server
<rpc>
  <lock-configuration/>
</rpc>
                       <rpc-reply xmlns:junos="URL"></rpc-reply>
```

Automatically Discard Uncommitted Changes

By default, if a client application locks the configuration and does not commit it before the JUNOScript session ends, any uncommitted changes that are made during the session are retained in the candidate configuration. When the candidate configuration is subsequently committed, the leftover changes become part of the committed configuration. This can lead to unexpected results if the user or application performing the subsequent commit is unaware of the leftover changes.

To discard uncommitted changes from the candidate configuration when the JUNOScript session ends before the client application commits the configuration, enclose the <rollback> tag element within the <lock-configuration> tag element and set the <rollback> tag element's value to automatic.

The following example illustrates the sequence of tag elements that causes an automatic rollback:



Client Application **JUNOScript Server**

```
<rpc>
  <lock-configuration>
    <rollback>automatic</rollback>
  </lock-configuration>
</rpc>
                              <rpc-reply xmlns:junos="URL"></rpc-reply>
```

Request Configuration Information

To request and parse configuration information, a client application emits the <get-configuration> tag element enclosed in an <rpc> tag element. By setting optional attributes on the opening <get-configuration> tag and enclosing the appropriate child tags within the <get-configuration> tag element, the client application can request either the candidate or the committed configuration, specify either JUNOScript-tagged or formatted ASCII output, and request the entire configuration or just a section of it. The following sections describe the procedures for requesting configuration information:

- Specify the Committed or Candidate Configuration on page 52
- Specify Formatted ASCII or JUNOScript-Tagged Output on page 52
- Specify Output Format for Inherited Configuration Groups on page 54
- Request the Complete Configuration on page 56
- Request One Hierarchy Level on page 57
- Request a Single Configuration Object on page 58

If the client application has locked the candidate configuration as described in "Lock the Candidate Configuration" on page 50, it should unlock it after making its read requests. Other users and applications cannot change the configuration while it remains locked. For more information, see "Unlock the Candidate Configuration" on page 77.

Client applications can also request an XML schema representation of the complete hierarchy of JUNOS configuration statements, as described in the following section:

- Request an XML Schema for the Configuration Hierarchy on page 59

Specify the Committed or Candidate Configuration

To request information from the current committed configuration—the one active on the router—set the database attribute on the opening <get-configuration> tag to the value committed. To request information from the current candidate configuration, either set the database attribute to the value candidate or omit the database attribute completely (the JUNOScript server returns information from the candidate configuration by default). In either case, enclose the <get-configuration> tag element in an <rpc> tag element.

The database attribute can be combined in the opening <get-configuration> tag with the format attribute (described in "Specify Formatted ASCII or JUNOScript-Tagged Output" on page 52), the inherit attribute (described in "Specify Output Format for Inherited Configuration Groups" on page 54), or both. It can be included when requesting either the entire configuration or sections of it (as described in "Request the Complete Configuration" on page 56, "Request One Hierarchy Level" on page 57, and "Request a Single Configuration Object" on page 58).

The following example uses the database attribute to request the entire committed configuration:

```
Client Application                      JUNOScript Server
<rpc>
  <get-configuration database="committed"/>
</rpc>
                                        <rpc-reply xmlns:junos="URL">
                                          <configuration>
                                            <version>5.6R1</version>
                                              <system>
                                                <host-name>big-router</host-name>
                                                <!- - other children of the <system> tag - ->
                                              </system>
                                            <!- - other children of the <configuration> tag - ->
                                          </configuration>
                                        </rpc-reply>
                                                                                    T1050
```

Specify Formatted ASCII or JUNOScript-Tagged Output

To request that the JUNOScript server return configuration data as formatted ASCII text rather than tagged with JUNOScript tag elements, set the format attribute on the opening <get-configuration> tag to the value text. The server formats its response in the same way as the JUNOS CLI show configuration command displays configuration data—it uses the newline character, tabs, braces, and square brackets to indicate the hierarchical relationships between configuration statements. The server encloses formatted ASCII configuration information in a <configuration-text> tag element.

To request JUNOScript-tagged output, either set the format attribute to the value xml or omit the format attribute completely (the JUNOScript server returns JUNOScript-tagged output by default). The JUNOScript server encloses its output in a <configuration> tag element.

When the JUNOScript server encloses a JUNOScript tag element in the <undocumented> tag element, the corresponding configuration element (hierarchy level or object) is not documented in the JUNOS software configuration guides or officially supported by Juniper Networks. Most often, the undocumented element is used for debugging purposes only by Juniper Networks personnel. In rarer cases, the element is no longer supported or has been moved to another area of the configuration hierarchy, but appears in the current location for backward compatibility.

Regardless of whether they are requesting JUNOScript tag elements or formatted ASCII, client applications must use JUNOScript tag elements to represent the configuration element to display. The format attribute controls the format of only the JUNOScript server's output. Enclose the tag elements that represent the configuration element to display in <get-configuration> and <rpc> tag elements.

The format attribute can be combined in the opening <get-configuration> tag with the database attribute (described in "Specify the Committed or Candidate Configuration" on page 52), the inherit attribute (described in "Specify Output Format for Inherited Configuration Groups" on page 54), or both. It can be included when requesting either the entire configuration or sections of it (as described in "Request the Complete Configuration" on page 56, "Request One Hierarchy Level" on page 57, and "Request a Single Configuration Object" on page 58).

The following example uses the format attribute to request ASCII-formatted output at the [edit policy-options] level of the current candidate configuration:

Client Application

```
<rpc>
   <get-configuration format="text">
      <configuration>
         <policy-options/>
      </configuration>
   </get-configuration>
</rpc>
```

JUNOScript Server

```
<rpc-reply xmlns:junos="URL">
   <configuration-text>
      policy-options {
         policy-statement load-balancing-policy {
            from {
               route-filter 192.168.10/24 orlonger;
               route-filter 9.114/16 orlonger;
            }
            then {
               load-balance per-packet;
            }
         }
      }
   </configuration-text>
</rpc-reply>
```

T1019

Specify Output Format for Inherited Configuration Groups

The <groups> tag element corresponds to the [edit groups] configuration hierarchy and encloses tag elements representing *configuration groups*, which define sets of configuration statements that can be inserted in multiple locations in the configuration hierarchy by using the apply-groups configuration statement. The section of configuration hierarchy to which a configuration group is applied is said to inherit the group's statements. For more information about configuration groups, see the *JUNOS Internet Software Configuration Guide: Getting Started*.

By default, the JUNOScript server displays the tag element for each configuration group as a child of the <groups> tag element, instead of indicating the inheritance relationships by displaying the tag elements defined in a configuration group as children of the inheriting tag elements. (This parallels the default behavior of the CLI configuration mode show command, which similarly displays the [edit groups] hierarchy as a separate hierarchy in the configuration.)

To request that the JUNOScript server enclose tag elements that are inherited from configuration groups within the tag elements that are inheriting them, and not display the <groups> tag element, set the inherit attribute on the opening <get-configuration> tag to the value inherit. The output is similar to the output from the following CLI configuration mode command:

```
user@host# show | display inheritance | except ##
```

The JUNOScript server encloses its output in <configuration> and <rpc-reply> tag elements.

The inherit attribute can be combined in the opening <get-configuration> tag with the database attribute (described in "Specify the Committed or Candidate Configuration" on page 52), the format attribute (described in "Specify Formatted ASCII or JUNOScript-Tagged Output" on page 52), or both. It can be included when requesting either the entire configuration or sections of it (as described in "Request the Complete Configuration" on page 56, "Request One Hierarchy Level" on page 57, and "Request a Single Configuration Object" on page 58).

For example, consider the following configuration hierarchy, which defines a configuration group called interface-group inherited by the [edit interfaces] hierarchy level:

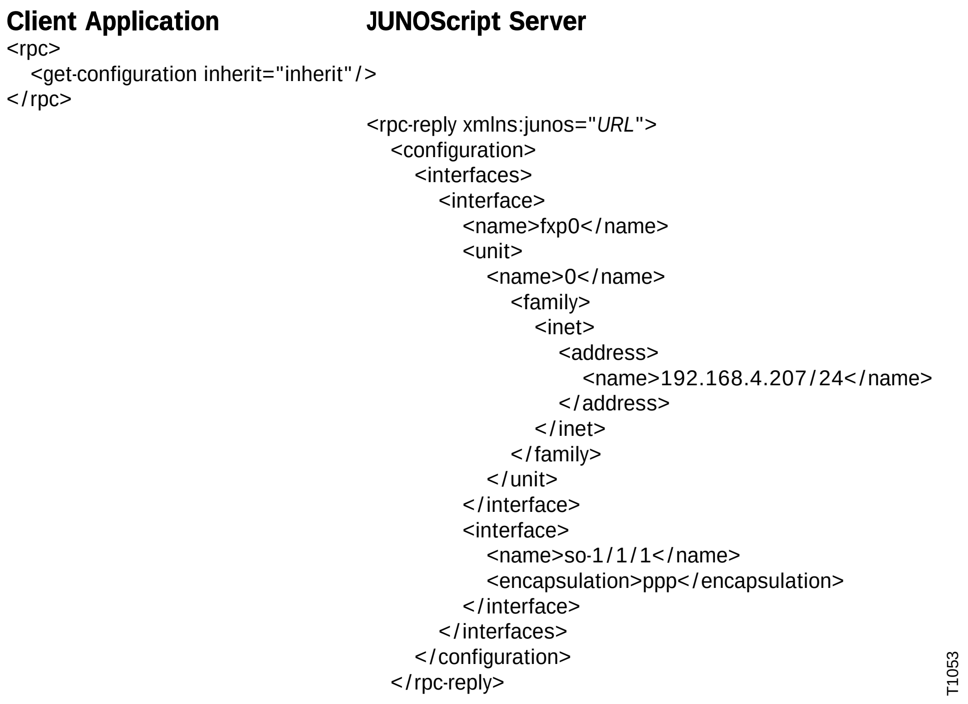
```
[edit]
groups {
  interface-group {
      interfaces {
        so-1/1/1 {
          encapsulation ppp;
        }
      }
  }
}
apply-groups interface-group;
interfaces {
  fxp0 {
    unit 0 {
      family inet {
        address 192.168.4.207/24;
      }
    }
  }
}
```

When the inherit attribute is not used, the output includes the <groups> tag element enclosing the tag elements defined in the interface-group configuration group. The placement of the <apply-groups> tag element directly above the <interfaces> tag element indicates that the [edit interfaces] hierarchy inherits the statements defined in the interface-group configuration group.

```
Client Application   JUNOScript Server
<rpc>
  <get-configuration/>
</rpc>
                     <rpc-reply xmlns:junos="URL">
                       <configuration>
                         <groups>
                           <name>interface-group</name>
                           <interfaces>
                             <interface>
                               <name>so-1/1/1</name>
                               <encapsulation>ppp</encapsulation>
                             </interface>
                           </interfaces>
                         </groups>
                         <apply-groups>interface-group</apply-groups>
                         <interfaces>
                           <interface>
                             <name>fxp0</name>
                             <unit>
                               <name>0</name>
                               <family>
                                 <inet>
                                   <address>
                                     <name>192.168.4.207/24</name>
                                   </address>
                                 </inet>
                               </family>
                             </unit>
                           </interface>
                         </interfaces>
                       </configuration>
                     </rpc-reply>
```

T1052

When the inherit attribute is used, the output includes the <interfaces> tag element enclosing the tag elements defined by the interface-group configuration group. The <groups> and <apply-groups> tag elements are not displayed.



Client Application **JUNOScript Server**

```
<rpc>
  <get-configuration inherit="inherit"/>
</rpc>
                              <rpc-reply xmlns:junos="URL">
                                <configuration>
                                  <interfaces>
                                    <interface>
                                      <name>fxp0</name>
                                      <unit>
                                        <name>0</name>
                                          <family>
                                            <inet>
                                              <address>
                                                <name>192.168.4.207/24</name>
                                              </address>
                                            </inet>
                                          </family>
                                        </unit>
                                      </interface>
                                      <interface>
                                        <name>so-1/1/1</name>
                                        <encapsulation>ppp</encapsulation>
                                      </interface>
                                    </interfaces>
                                  </configuration>
                                </rpc-reply>
```

T1053

Request the Complete Configuration

To request the entire current committed or candidate configuration, emit the empty <get-configuration/> tag enclosed in an <rpc> tag element. If desired, include any of the following optional attributes on the <get-configuration/> tag:

- The database attribute, as described in “Specify the Committed or Candidate Configuration” on page 52
- The format attribute, as described in “Specify Formatted ASCII or JUNOScript-Tagged Output” on page 52
- The inherit attribute, as described in “Specify Output Format for Inherited Configuration Groups” on page 54

The following example illustrates the tag sequence for requesting the current candidate configuration tagged with JUNOScript tag elements (the default):

```
Client Application    JUNOScript Server
<rpc>
  <get-configuration/>
</rpc>
                      <rpc-reply xmlns:junos="URL">
                        <configuration>
                          <version>5.6R1</version>
                          <system>
                            <host-name>big-router</host-name>
                            <!- - other children of the <system> tag - ->
                          </system>
                          <!- - other children of the <configuration> tag - ->
                        </configuration>
                      </rpc-reply>
```

T1051

Request One Hierarchy Level

To request a single hierarchy level from the current committed or candidate configuration, emit a <get-configuration> tag element and enclose the tag elements representing all levels of the configuration hierarchy from the root (represented by the <configuration> tag element) down to the level to display. Use an empty tag to represent the requested level. Enclose the entire request in an <rpc> tag element.

If desired, include any of the following optional attributes on the opening <get-configuration> tag:

- The database attribute, as described in "Specify the Committed or Candidate Configuration" on page 52
- The format attribute, as described in "Specify Formatted ASCII or JUNOScript-Tagged Output" on page 52

> **Note** The format attribute controls the format of only the JUNOScript server's output. Client applications must emit JUNOScript tag elements instead of formatted ASCII to represent which configuration level to display.

- The inherit attribute, as described in "Specify Output Format for Inherited Configuration Groups" on page 54

The following example illustrates the tag sequence when a client application requests the contents of the [edit system login] level of the current candidate configuration. The output is tagged with JUNOScript tag elements (the default).

```
Client Application    JUNOScript Server
<rpc>
  <get-configuration>
    <configuration>
      <system>
        <login/>
      </system>
    </configuration>
  </get-configuration>
</rpc>
                      <rpc-reply xmlns:junos="URL">
                        <configuration>
                          <system>
                            <login>
                              <user>
                                <name>barbara</name>
                                <full-name>Barbara Anderson</full-name>
                                <!- - other child tags for this user - ->
                              </user>
                              <!- - other children of the <login> tag - ->
                            </login>
                          </system>
                        </configuration>
                      </rpc-reply>
```

T1021

Request a Single Configuration Object

To request information about a single object at a specific level of the configuration hierarchy, emit a <get-configuration> tag element enclosing the tag elements representing the entire configuration hierarchy down to the object to display. To represent the requested object, emit its container tag element and identifier tag element only, not any tag elements that represent other characteristics. Enclose the entire request in an <rpc> tag element.

If desired, include any of the following optional attributes on the opening <get-configuration> tag:

- The database attribute, as described in “Specify the Committed or Candidate Configuration” on page 52
- The format attribute, as described in “Specify Formatted ASCII or JUNOScript-Tagged Output” on page 52

> **Note**
> The format attribute controls the format of only the JUNOScript server’s output. Client applications must emit JUNOScript tag elements instead of formatted ASCII to represent which configuration level to display.

- The inherit attribute, as described in “Specify Output Format for Inherited Configuration Groups” on page 54

The following example illustrates the tag sequence when a client application requests the contents of one multicasting scope called local, which is at the [edit routing-options multicast] hierarchy level. To specify the configuration object about which to supply information, the client application emits the <name>local</name> identifier tag element as the innermost tag element. The output is from the current candidate configuration and is tagged with JUNOScript tag elements (the default).

```
Client Application                JUNOScript Server
<rpc>
  <get-configuration>
    <configuration>
      <routing-options>
        <multicast>
          <scope>
            <name>local</name>
          </scope>
        </multicast>
      </routing-options>
    </configuration>
  </get-configuration>
</rpc>
                                  <rpc-reply xmlns:junos="URL">
                                    <configuration>
                                      <routing-options>
                                        <multicast>
                                          <scope>
                                            <name>local</name>
                                            <prefix>239.255.0.0/16</prefix>
                                            <interface>ip-f/p/0</interface>
                                          </scope>
                                        </multicast>
                                      </routing-options>
                                    </configuration>
                                  </rpc-reply>
```

T1022

Request an XML Schema for the Configuration Hierarchy

To request an XML Schema-language representation of the entire JUNOS configuration hierarchy, emit the <get-xnm-information> tag element enclosing the following two child tag elements with the indicated values:

- <type>xml-schema</type>, which specifies that the JUNOScript server return the configuration as an XML schema
- <namespace>junos-configuration</namespace>, which specifies that the JUNOScript server return information about the JUNOS configuration

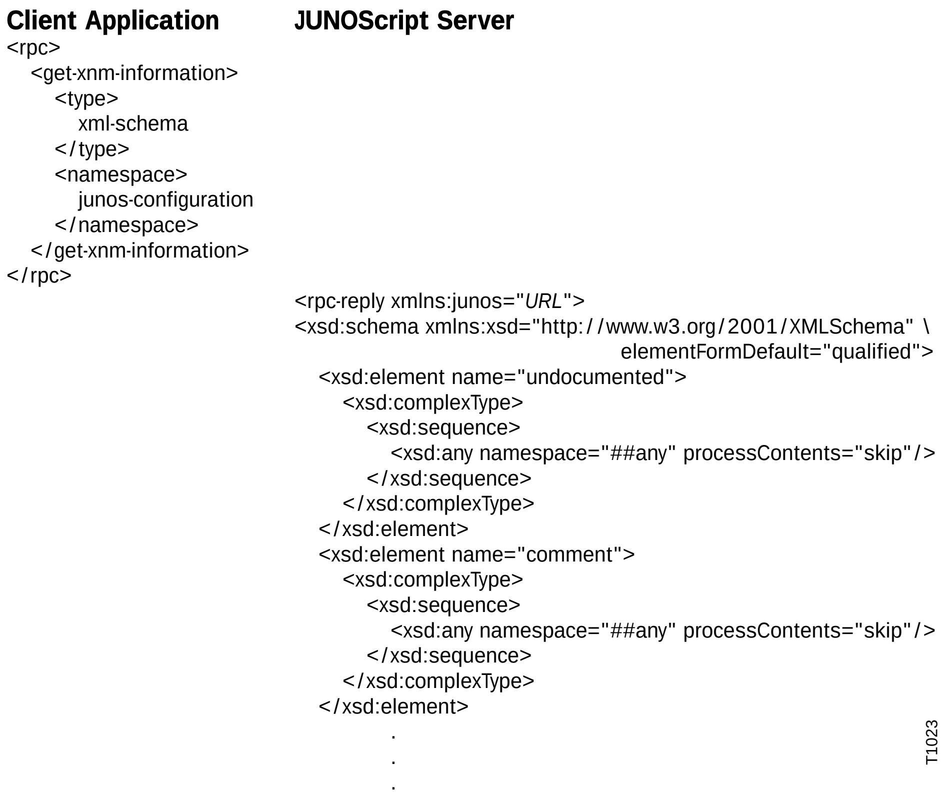
Enclose the entire request in an <rpc> tag element.

The JUNOScript server returns the XML schema enclosed in <rpc-reply> and <xsd:schema> tag elements. The JUNOScript configuration schema represents the entire set of elements that can be configured on a Juniper Networks router that is running the version of the JUNOS software specified by the xmlns:junos attribute in the opening <rpc-reply> tag. Client applications can use the schema to validate the candidate or committed schema on a router, or to discover which configuration statements are available in that version of the JUNOS software.

The JUNOScript configuration schema does not indicate which elements are actually configured on the router where the JUNOScript server is running, or even that an element can be configured on that type of router (some configuration statements are available only on certain router types). To request the set of currently configured elements and their settings, emit the <get-configuration> tag element instead, as described in other subsections of "Request Configuration Information" on page 51.

Explaining the structure and notational conventions of the XML Schema language is beyond the scope of this document. For a basic introduction to the XML Schema language, see the primer available at http://www.w3.org/TR/xmlschema-0. The primer references the more formal specifications for the XML Schema language if you need additional details.

The following examples illustrate the tag sequence with which a client application requests the JUNOScript configuration schema. In the JUNOScript server's reply, the first two <xsd:element> statements define the schema for the <undocumented> and <comment> JUNOScript tag elements, which can be enclosed in most other container tag elements defined in the schema (container tag elements are defined as <xsd:complexType>).



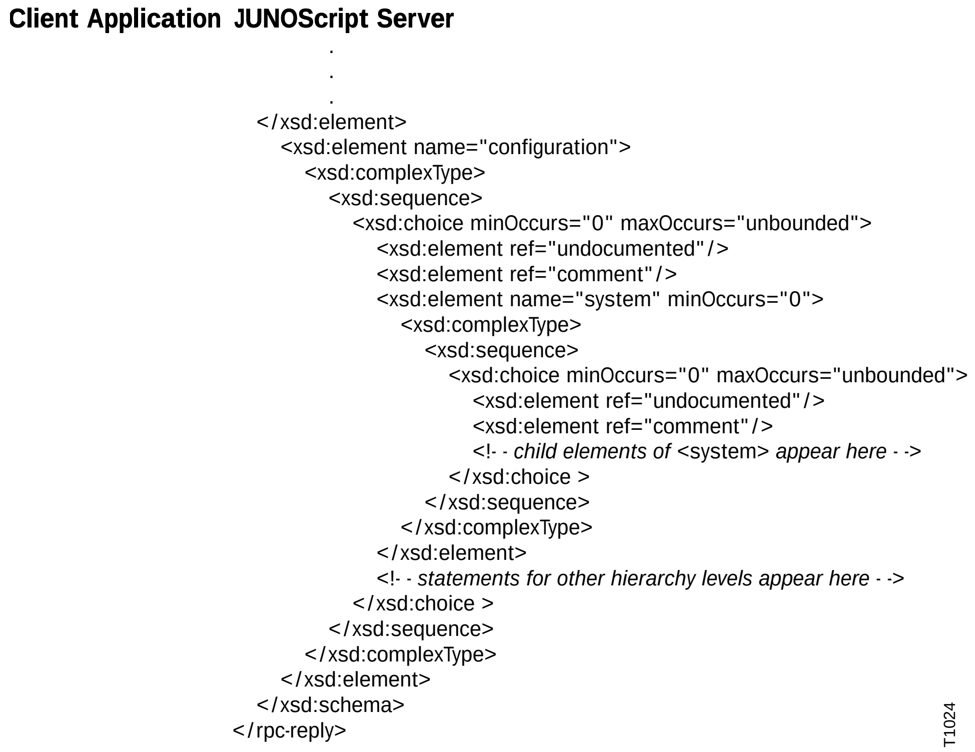
Client Application

```
<rpc>
  <get-xnm-information>
    <type>
      xml-schema
    </type>
    <namespace>
      junos-configuration
    </namespace>
  </get-xnm-information>
</rpc>
```

JUNOScript Server

```
<rpc-reply xmlns:junos="URL">
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" \
                                        elementFormDefault="qualified">
  <xsd:element name="undocumented">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any namespace="##any" processContents="skip"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="comment">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any namespace="##any" processContents="skip"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
          .
          .
          .
```

The third <xsd:element> statement in the schema defines the JUNOScript <configuration> tag element. The fourth <xsd:element> statement begins the definition of the <system> tag element, which corresponds to the [edit system] level of the JUNOScript configuration hierarchy. The statements corresponding to other hierarchy levels are omitted for brevity.



```
                .
                .
                .
        </xsd:element>
          <xsd:element name="configuration">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:choice minOccurs="0" maxOccurs="unbounded">
                  <xsd:element ref="undocumented"/>
                  <xsd:element ref="comment"/>
                  <xsd:element name="system" minOccurs="0">
                    <xsd:complexType>
                      <xsd:sequence>
                        <xsd:choice minOccurs="0" maxOccurs="unbounded">
                          <xsd:element ref="undocumented"/>
                          <xsd:element ref="comment"/>
                          <!-- child elements of <system> appear here -->
                        </xsd:choice >
                      </xsd:sequence>
                    </xsd:complexType>
                  </xsd:element>
                  <!-- statements for other hierarchy levels appear here -->
                </xsd:choice >
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:schema>
      </rpc-reply>
```

Change the Candidate Configuration

To change the current candidate configuration, emit the <load-configuration> tag element enclosed in an <rpc> tag element. Specify which configuration element (hierarchy level or configuration object) to change in one of two ways:

- By setting the empty <load-configuration/> tag's url attribute to the pathname of a file that resides on the router and contains the set of configuration statements to load. For example, the following tag identifies the file /tmp/new.conf as the file to load:

  ```
  <load-configuration url="/tmp/new.conf"/>
  ```

- By enclosing within the <load-configuration> tag element the tag elements that represent the configuration statements to load. Include tag elements for the complete statement path down to the element to change.

The JUNOScript server encloses its response in <rpc-reply> and <load-configuration-results> tag elements. If the load operation succeeds, the <load-configuration-results> tag element encloses the <load-success/> tag. If the load operation fails, the <load-configuration-results> tag element encloses the <load-error-count> tag element, which indicates the number of errors that occurred. In this case, eliminate the errors before committing the candidate configuration.

Client applications can provide the configuration information to load either as formatted ASCII or tagged with JUNOScript tag elements. They can also specify the manner in which the JUNOScript server loads the configuration. For instructions, see the following sections:

- Provide Configuration Data as Formatted ASCII or JUNOScript Tag Elements on page 62
- Merge Statements into the Current Configuration on page 63
- Replace (Override) the Entire Current Configuration on page 64
- Replace a Configuration Element on page 64
- Delete a Configuration Element on page 66
- Change a Configuration Element's Activation State on page 69
- Replace a Configuration Element and Change Its Activation State Simultaneously on page 70
- Roll Back to a Previous Configuration on page 71

Provide Configuration Data as Formatted ASCII or JUNOScript Tag Elements

Client applications can use one of two formats when providing configuration data to be loaded into the candidate configuration:

- If providing formatted ASCII (the standard format used by the JUNOS CLI show configuration command to display configuration data), set the format attribute on the opening <load-configuration> tag to the value text. Enclose the configuration data in a <configuration-text> tag element rather than a <configuration> tag element. To indicate the hierarchical relationships between the configuration statements to be loaded, format them using the newline character, tabs and other white space, braces, and square brackets.

- If providing JUNOScript-tagged information, either set the format attribute on the opening <load-configuration> tag to the value xml or omit the format attribute completely (the JUNOScript server expects JUNOScript-tagged output by default). Enclose the configuration information in a <configuration> tag element.

Whichever form of configuration information is provided, enclose the <load-configuration> tag element in an <rpc> tag element. The format attribute can be combined with the action attribute, described in "Merge Statements into the Current Configuration" on page 63, "Replace (Override) the Entire Current Configuration" on page 64, and "Replace a Configuration Element" on page 64. For examples of the possible combinations, see those sections.

Merge Statements into the Current Configuration

To combine the statements in the loaded configuration with the current candidate configuration, set the action attribute on the opening <load-configuration> tag to the value merge. This is also the default behavior if there is no action attribute.

```
<load-configuration action="merge">
```

Merging configuration statements is useful when adding a new configuration object or subhierarchy to the configuration. If statements in the loaded configuration conflict with statements in the current candidate configuration, the loaded statements replace the current ones.

As noted in “Provide Configuration Data as Formatted ASCII or JUNOScript Tag Elements” on page 62, client applications can specify the configuration information to load either as formatted ASCII or tagged with JUNOScript tag elements. In the former case, set the format attribute on the opening <load-configuration> tag to text.

The following example merges information for a new interface called so-3/0/0 into the [edit interfaces] level of the current candidate configuration. The information is provided in JUNOScript-tagged format (the default).

Client Application

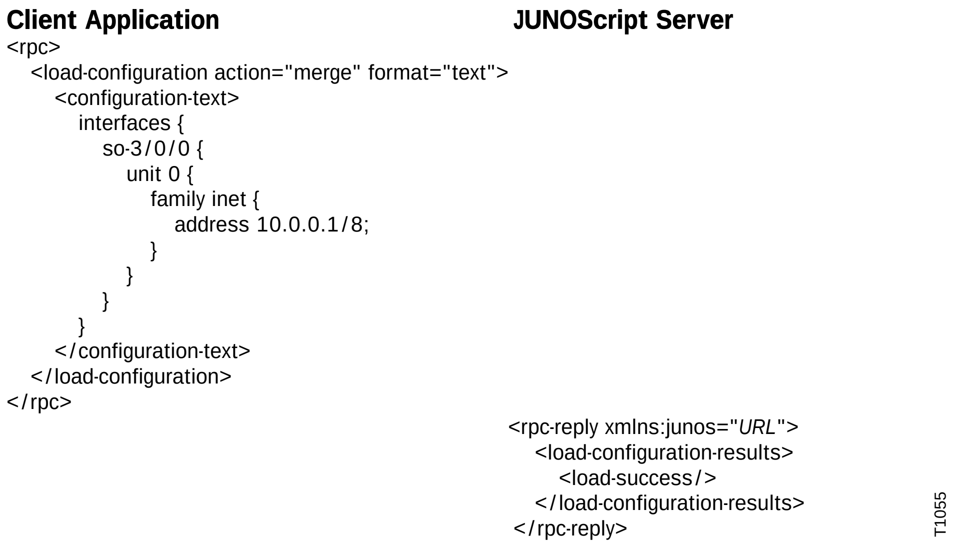
```
<rpc>
  <load-configuration action="merge">
    <configuration>
      <interfaces>
        <interface>
          <name>so-3/0/0</name>
          <unit>
            <name>0</name>
            <family>
              <inet>
                <address>
                  <name>10.0.0.1/8</name>
                </address>
              </inet>
            </family>
          </unit>
        </interface>
      </interfaces>
    </configuration>
  </load-configuration>
</rpc>
```

JUNOScript Server

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1054

The following example uses formatted ASCII to define the same new interface:



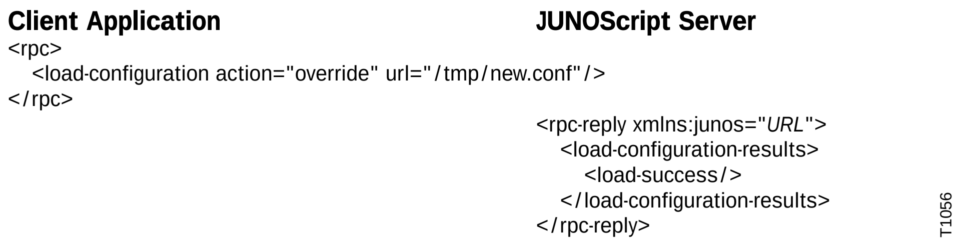
```
Client Application                                 JUNOScript Server
<rpc>
  <load-configuration action="merge" format="text">
    <configuration-text>
      interfaces {
        so-3/0/0 {
          unit 0 {
            family inet {
              address 10.0.0.1/8;
            }
          }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>
                                          <rpc-reply xmlns:junos="URL">
                                            <load-configuration-results>
                                              <load-success/>
                                            </load-configuration-results>
                                          </rpc-reply>
```

Replace (Override) the Entire Current Configuration

To discard the entire current candidate configuration and replace it with the loaded configuration, set the action attribute on the opening <load-configuration> tag to the value override:

```
<load-configuration action="override">
```

In the following example, the contents of the file /tmp/new.conf (which resides on the router) replaces the entire current candidate configuration. The information in the file is tagged with JUNOScript tag elements (the default), so the format attribute is not set.



```
Client Application                                 JUNOScript Server
<rpc>
  <load-configuration action="override" url="/tmp/new.conf"/>
</rpc>
                                          <rpc-reply xmlns:junos="URL">
                                            <load-configuration-results>
                                              <load-success/>
                                            </load-configuration-results>
                                          </rpc-reply>
```

Replace a Configuration Element

To replace a configuration element (hierarchy level or configuration object) in the current configuration, set the action attribute on the opening <load-configuration> tag to the value replace:

```
<load-configuration action="replace">
```

If using JUNOScript tag elements to define the loaded configuration, include the tag elements that represent the entire hierarchy down to the configuration element you want to replace. In the opening tag of the container tag element that represents the configuration element, set the replace attribute to the value replace. Within the container tag element, include all its child tag elements.

If using formatted ASCII to define the loaded configuration, include the statements that represent the entire hierarchy down to the element you want to replace. Place the replace: statement above the element's parent statement. Within the container tag elements for the element, include all relevant child statements.

The following example grants new permissions for the object named operator at the [edit system login class] hierarchy level. The information is provided in JUNOScript-tagged format (the default).

```
Client Application                              JUNOScript Server
<rpc>
  <load-configuration action="replace">
    <configuration>
      <system>
        <login>
          <class replace="replace">
            <name>operator</name>
            <permissions>configure</permissions>
            <permissions>admin-control</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
                                                <rpc-reply xmlns:junos="URL">
                                                  <load-configuration-results>
                                                    <load-success/>
                                                  </load-configuration-results>
                                                </rpc-reply>
```

T1057

The following example uses formatted ASCII to make the same change:

```
Client Application                              JUNOScript Server
<rpc>
  <load-configuration action="replace" format="text">
    <configuration-text>
      system {
        login {
        replace:
          class operator {
            permissions [ configure admin-control ];
          }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>
                                                <rpc-reply xmlns:junos="URL">
                                                  <load-configuration-results>
                                                    <load-success/>
                                                  </load-configuration-results>
                                                </rpc-reply>
```

T1058

Delete a Configuration Element

The client application can use the delete attribute to delete several kinds of configuration elements:

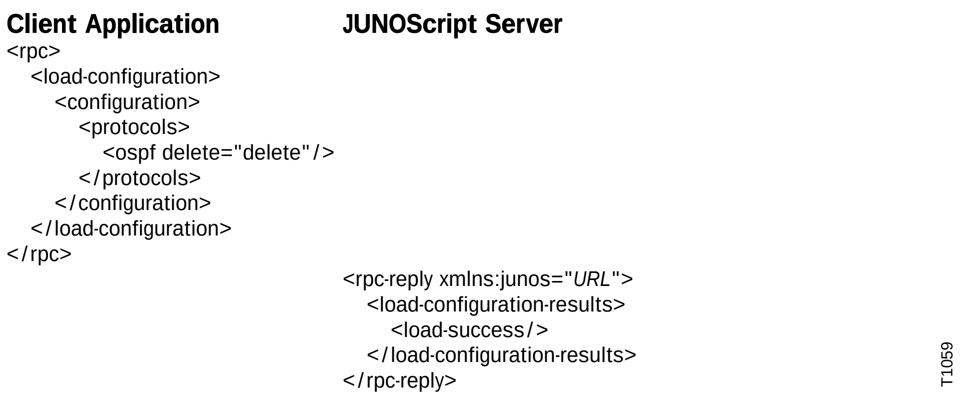
- Delete a Hierarchy Level on page 66
- Delete a Configuration Object on page 67
- Delete One or More Values from a Leaf Statement on page 68
- Delete a Fixed-Form Option on page 68

Note The JUNOS CLI does not provide a delete: statement that marks formatted ASCII configuration data for deletion. Client applications must use JUNOScript tag elements to represent the data to delete.

Delete a Hierarchy Level

To remove a hierarchy level from the configuration hierarchy, set the delete attribute to the value delete on the empty tag that represents the level. Emit a <load-configuration> tag element enclosing the tag elements representing the entire statement path down to the level to remove.

The following example removes the [edit protocols ospf] level of the current candidate configuration:



Client Application **JUNOScript Server**

```
<rpc>
    <load-configuration>
        <configuration>
            <protocols>
                <ospf delete="delete"/>
            </protocols>
        </configuration>
    </load-configuration>
</rpc>
```

```
<rpc-reply xmlns:junos="URL">
    <load-configuration-results>
        <load-success/>
    </load-configuration-results>
</rpc-reply>
```

T1059

Delete a Configuration Object

To delete a single configuration object, set the delete attribute to the value delete on the container tag element for that object. Inside the container tag element, include the identifier tag element only, not any tag elements that represent other characteristics. Emit a <load-configuration> tag enclosing the tag elements representing the entire statement path down to the object to remove.

Note

The delete attribute is placed on the opening tag of the containing tag element rather than on the identifier tag element (in the following example, on the <user> tag element rather than the <name> tag element). The presence of the identifier tag element results in the removal of the specified object rather than the removal of the entire hierarchy level represented by the containing tag element (in the example, the user account barbara is removed rather than the entire [edit system login user] level).

The following example removes the account for user object barbara from the [edit system login user] level of the current candidate configuration:

Client Application

```
<rpc>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <user delete="delete">
            <name>barbara</name>
          </user>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
```

JUNOScript Server

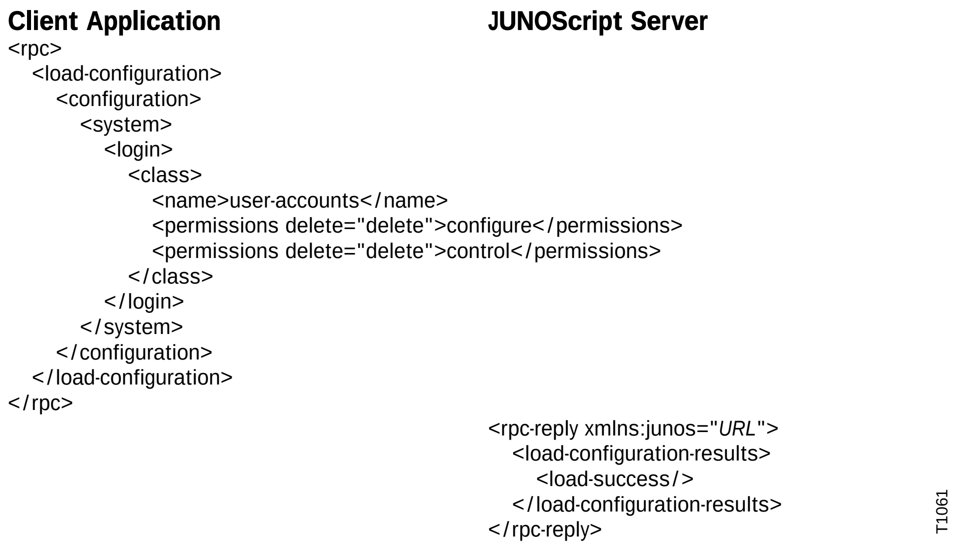
```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1060

Delete One or More Values from a Leaf Statement

To delete one or more values from a leaf statement that accepts multiple values, set the delete attribute to the value delete on the opening tag for each value. Do not include tag elements that represent values to be retained. Emit a <load-configuration> tag element enclosing the tag elements representing the entire statement path down to the leaf statement from which to remove values.

The following example removes two of the permissions granted to the user-accounts login class:

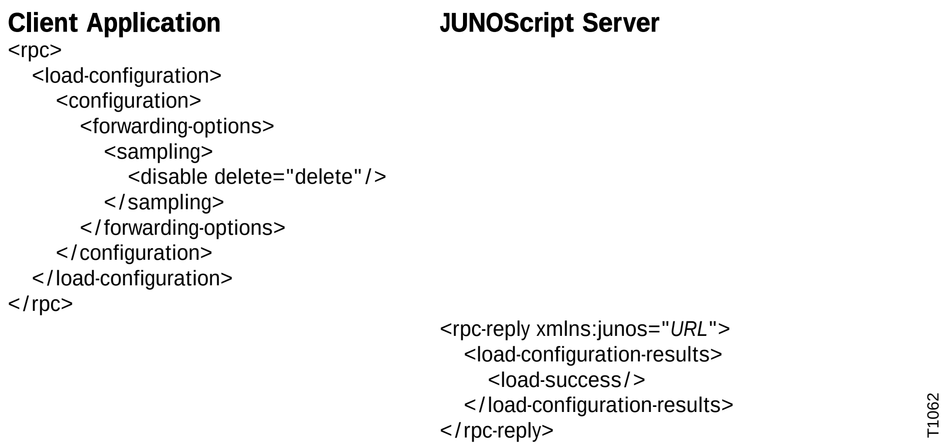


```
Client Application                         JUNOScript Server
<rpc>
  <load-configuration>
    <configuration>
      <system>
        <login>
          <class>
            <name>user-accounts</name>
            <permissions delete="delete">configure</permissions>
            <permissions delete="delete">control</permissions>
          </class>
        </login>
      </system>
    </configuration>
  </load-configuration>
</rpc>
                                           <rpc-reply xmlns:junos="URL">
                                             <load-configuration-results>
                                               <load-success/>
                                             </load-configuration-results>
                                           </rpc-reply>
```
T1061

Delete a Fixed-Form Option

To delete a fixed-form option, set the delete attribute to the value delete on the tag element for the option. Emit a <load-configuration> tag element enclosing the tag elements that represent the entire statement path down to the option to remove.

The following example removes the fixed-form disable option at the [edit forwarding-options sampling] hierarchy level:



```
Client Application                     JUNOScript Server
<rpc>
  <load-configuration>
    <configuration>
      <forwarding-options>
        <sampling>
          <disable delete="delete"/>
        </sampling>
      </forwarding-options>
    </configuration>
  </load-configuration>
</rpc>
                                       <rpc-reply xmlns:junos="URL">
                                         <load-configuration-results>
                                           <load-success/>
                                         </load-configuration-results>
                                       </rpc-reply>
```
T1062

Change a Configuration Element's Activation State

When a configuration element (hierarchy level or configuration object) is deactivated in the configuration hierarchy, it remains in the candidate configuration but is not activated in the actual configuration when the candidate is later committed.

To use JUNOScript tag elements to define which element to deactivate, either omit the format attribute from the opening <load-configuration> tag or set it to the value xml. Emit the opening <configuration> tag next, and then the tag elements representing the entire statement path down to the element to deactivate. On the tag element that represents the configuration element itself, set the inactive attribute to the value inactive:

- To represent a hierarchy level, emit an empty tag.
- To represent an object, emit the object's container tag element. Inside the container tag element enclose only the identifier tag element for the object, not any tag elements that represent other object attributes.

Conclude the tag stream with closing </configuration> and </load-configuration> tags.

To use formatted ASCII to define the configuration element to deactivate, set the format attribute on the opening <load-configuration> tag to the value text. Emit the opening <configuration-text> tag element next, followed by formatted ASCII for all statements in the path down to the element you want to deactivate. Place the inactive: statement immediately before the statement for the element. Conclude the tag stream with closing </configuration-text> and </load-configuration> tags.

To reactivate a configuration element that was previously deactivated, use the preceding instructions and substitute the string active for inactive. Specifically, when loading JUNOScript tag elements, set the active attribute to the value active on the opening tag for the element to activate. When loading formatted ASCII statements, place the active: statement immediately before the statements to reactivate. With both kinds of notation, the reactivated element is activated in the committed configuration the next time the candidate configuration is committed.

The following example deactivates the [edit protocols isis] level of the current candidate configuration:

```
Client Application                JUNOScript Server
<rpc>
  <load-configuration>
    <configuration>
      <protocols>
        <isis inactive="inactive"/>
      </protocols>
    </configuration>
  </load-configuration>
</rpc>
                                  <rpc-reply xmlns:junos="URL">
                                    <load-configuration-results>
                                      <load-success/>
                                    </load-configuration-results>
                                  </rpc-reply>
```

T1063

Replace a Configuration Element and Change Its Activation State Simultaneously

To replace a configuration element (hierarchy level or configuration object) completely while simultaneously deactivating or reactivating it, set the action attribute on the opening <load-configuration> tag to the value replace. Within the container tag elements for the configuration element you are replacing, include the tag elements or statements that represent all the element's characteristics, not just its identifier tag element or statement. Finally, combine attributes or statements as follows:

- If using JUNOScript tag elements to replace and deactivate an element, set two attributes on the opening tag of its container tag element: the replace attribute to the value replace and the inactive attribute to the value inactive. If using formatted ASCII, place the replace: statement above the statements to replace and the inactive: statement directly in front of the first statement.

- If using JUNOScript tag elements to replace and reactivate an element, set two attributes on the opening tag of its container tag element: the replace attribute to the value replace and the active attribute to the value active. If using formatted ASCII, place the replace: statement above the statements to replace and the active: statement directly in front of the first statement.

For more information about completely reconfiguring an element, see "Replace a Configuration Element" on page 64.

The following example replaces the information in the [edit protocols bgp] level of the current candidate configuration for the group called G3, but also deactivates the group so that it is not activated in the actual configuration when the candidate is committed:

Client Application

```
<rpc>
  <load-configuration action="replace">
    <configuration>
      <protocols>
        <bgp>
          <group replace="replace" inactive="inactive">
            <name>G3</name>
            <type>external</type>
            <peer-as>58</peer-as>
            <neighbor>
              <name>10.0.20.1</name>
            </neighbor>
          </group>
        </bgp>
      </protocols>
    </configuration>
  </load-configuration>
</rpc>
```

JUNOScript Server

```
<rpc-reply xmlns:junos="URL">
  <load-configuration-results>
    <load-success/>
  </load-configuration-results>
</rpc-reply>
```

T1064

The following example uses formatted ASCII to make the same change:

```
Client Application                                  JUNOScript Server
<rpc>
  <load-configuration action="replace" format="text">
    <configuration-text>
      protocols {
        bgp {
        replace:
          inactive: group G3 {
            type external;
            peer-as 58;
            neighbor 10.0.20.1;
          }
        }
      }
    </configuration-text>
  </load-configuration>
</rpc>
                                                    <rpc-reply xmlns:junos="URL">
                                                      <load-configuration-results>
                                                        <load-success/>
                                                      </load-configuration-results>
                                                    </rpc-reply>
```

T1065

Roll Back to a Previous Configuration

The router stores a copy of the most recently committed configuration and up to nine previous configurations. To replace the current candidate configuration with a previous one, set the rollback attribute on the empty <load-configuration/> tag to the numerical index for the appropriate previous configuration. The index for the most recently committed configuration is 0 (zero), and the index for the oldest possible stored configuration is 9.

In the following example, the current candidate configuration is replaced by the most recently committed one:

```
Client Application                  JUNOScript Server
<rpc>
  <load-configuration rollback="0"/>
</rpc>
                                    <rpc-reply xmlns:junos="URL">
                                      <load-configuration-results>
                                        <load-success/>
                                      </load-configuration-results>
                                    </rpc-reply>
```

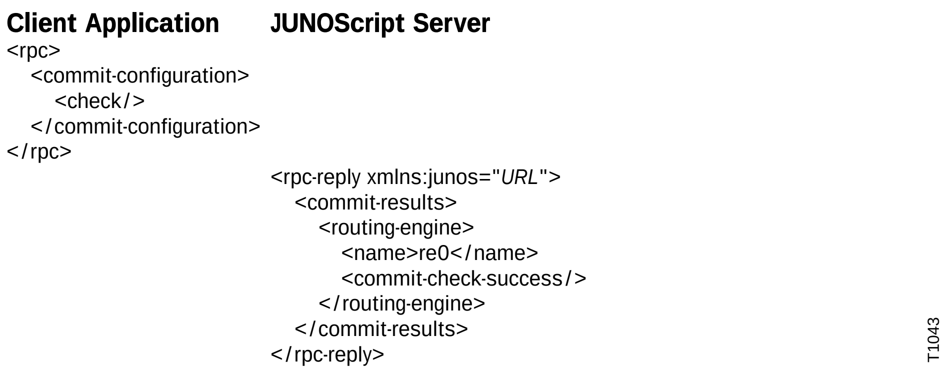
T1066

Verify the Syntactic Correctness of the Candidate Configuration

Before committing the candidate configuration, you might want to confirm that it is syntactically correct. To verify the candidate configuration without actually committing it, enclose the empty <check/> tag in a <commit-configuration> tag element.

The JUNOScript server encloses its response in <rpc-reply>, <commit-results>, and <routing-engine> tag elements. If the syntax check succeeds, the <routing-engine> tag element encloses the <commit-check-success/> tag and the <name> tag element, which reports the name of the Routing Engine on which the check succeeded (either re0 or re1). If the syntax check fails, an <xnm:error> tag element encloses tag elements that describe the error.

The following example illustrates the tag sequence when the candidate configuration on Routing Engine 0 is syntactically correct:



```
Client Application           JUNOScript Server
<rpc>
  <commit-configuration>
    <check/>
  </commit-configuration>
</rpc>
                             <rpc-reply xmlns:junos="URL">
                               <commit-results>
                                 <routing-engine>
                                   <name>re0</name>
                                   <commit-check-success/>
                                 </routing-engine>
                               </commit-results>
                             </rpc-reply>
```

T1043

Commit the Candidate Configuration

To commit the current candidate configuration so that it becomes the active configuration on the router, emit the empty <commit-configuration/> tag enclosed in an <rpc> tag element. To avoid inadvertently committing changes made by other users or applications, lock the candidate configuration before changing it and emit the <commit-configuration/> tag while the configuration is still locked. (For instructions on locking and changing the candidate configuration, see “Lock the Candidate Configuration” on page 50 and “Change the Candidate Configuration” on page 61.) After committing the configuration, unlock it as described in “Unlock the Candidate Configuration” on page 77.

The JUNOScript server encloses its response in <rpc-reply>, <commit-results>, and <routing-engine> tag elements. If the commit operation succeeds, the <routing-engine> tag element encloses the <commit-success/> tag and the <name> tag element, which reports the name of the Routing Engine on which the commit operation succeeded (either re0 or re1). If the commit operation fails, an <xnm:error> tag element encloses tag elements that describe the error. The most common causes of failure are semantic or syntactic errors in the candidate configuration.

The following example illustrates the tag sequence when a client application commits the candidate configuration on Routing Engine 0:

```
Client Application      JUNOScript Server
<rpc>
  <commit-configuration/>
</rpc>
                        <rpc-reply xmlns:junos="URL">
                          <commit-results>
                            <routing-engine>
                              <name>re0</name>
                              <commit-success/>
                            </routing-engine>
                          </commit-results>
                        </rpc-reply>
```

T1044

For information about using JUNOScript for other commit operations, see the following sections:

- Commit and Synchronize the Configuration on Both Routing Engines on page 73
- Commit the Configuration at a Specified Time on page 74
- Commit a Configuration but Require Confirmation on page 76

For more detailed information about commit operations, including a discussion of the interaction among different commit operations, see the *JUNOS Internet Software Configuration Guide: Getting Started*.

Commit and Synchronize the Configuration on Both Routing Engines

To copy the candidate configuration stored on one of a router's Routing Engines to the other Routing Engine, verify the candidate's syntactic correctness, and commit it on both Routing Engines, emit the empty <synchronize/> tag enclosed in <commit-configuration> and <rpc> tag elements. This operation is valid only on a router with more than one Routing Engine installed (the JUNOScript server returns an error if there is only one Routing Engine). Also, the apply groups re0 and re1 must already be defined on the router. For more information, see the *JUNOS Internet Software Configuration Guide: Getting Started*.

The JUNOScript server encloses its response in <rpc-reply> and <commit-results> tag elements. It emits a separate <routing-engine> tag element for each operation on each Routing Engine:

- If the syntax check succeeds on a Routing Engine, the <routing-engine> tag element encloses the <commit-check-success/> tag and the <name> tag element, which reports the name of the Routing Engine on which the check succeeded (either re0 or re1). If the configuration is incorrect, an <xnm:error> tag element encloses tag elements that describe the error.

- If the commit operation succeeds on a Routing Engine, the <routing-engine> tag element encloses the <commit-success/> tag and the <name> tag element, which reports the name of the Routing Engine on which the commit operation succeeded. If the commit operation fails, an <xnm:error> tag element encloses tag elements that describe the error. The most common causes of failure are semantic or syntactic errors in the candidate configuration.

The following example illustrates the tag sequence when a client application commits and synchronizes the candidate configuration:

```
Client Application      JUNOScript Server
<rpc>
  <commit-configuration>
    <synchronize/>
  </commit-configuration>
</rpc>
                        <rpc-reply xmlns:junos="URL">
                          <commit-results>
                            <routing-engine>
                              <name>re0</name>
                              <commit-check-success/>
                            </routing-engine>
                            <routing-engine>
                              <name>re1</name>
                              <commit-check-success/>
                            </routing-engine>
                            <routing-engine>
                              <name>re0</name>
                              <commit-success/>
                            </routing-engine>
                            <routing-engine>
                              <name>re1</name>
                              <commit-success/>
                            </routing-engine>
                          </commit-results>
                        </rpc-reply>
```

T1046

Commit the Configuration at a Specified Time

To commit the candidate configuration at a specified time in the future, emit the <at-time> tag element enclosed in <commit-configuration> and <rpc> tag elements. The configuration is checked immediately for syntactic correctness. If the syntax check succeeds, the JUNOScript server returns <rpc-reply>, <commit-results>, and <routing-engine> tag elements enclosing the <commit-check-success/> tag and the <name> tag element, which reports the name of the Routing Engine on which the check succeeded (either re0 or re1).

The configuration is scheduled for commit at the specified time. The JUNOScript server does not emit additional tag elements when the commit operation is actually performed.

If the configuration is not syntactically correct, an <xnm:error> tag element encloses tag elements that describe the error. The commit operation is not scheduled.

To indicate when to perform the commit operation, include one of three types of values in the <at-time> tag element:

- The string reboot, to commit the configuration the next time the router reboots.
- A time value of the form *hh*:*mm* [:*ss*] (hours, minutes, and optionally seconds), to commit the configuration at the specified time, which must be in the future at the time the client application emits the <commit-configuration> tag element but before 11:59:59 PM on the day it emits the tag element. As an example, if the <at-time> tag element specifies 02:00 AM and the application emits the <commit-configuration> tag element at 2:10 AM, the commit will not take place at all because the scheduled time has already passed for that day. The commit is not scheduled for 2:00 AM the next day.

  Use 24-hour time; for example, 04:30:00 means 4:30:00 AM and 20:00 means 8:00 PM. The time is interpreted with respect to the clock and time zone settings on the router.
- A date and time value of the form *yyyy-mm-dd hh*:*mm* [:*ss*] (year, month, date, hours, minutes, and optionally seconds), to commit the configuration at the specified day and time, which must be after the <commit-configuration> tag element is emitted. Use 24-hour time. For example, 2003-08-21 12:30:00 means 12:30 PM on August 21, 2003. The time is interpreted with respect to the clock and time zone settings on the router.

The following example illustrates the tag sequence when a client application schedules a commit operation for 10:00 PM on the current day:

Client Application

```
<rpc>
  <commit-configuration>
    <at-time>22:00</at-time>
  </commit-configuration>
</rpc>
```

JUNOScript Server

```
<rpc-reply xmlns:junos="URL">
  <commit-results>
    <routing-engine>
      <name>re1</name>
      <commit-check-success/>
    </routing-engine>
  </commit-results>
</rpc-reply>
```

T1047

Commit a Configuration but Require Confirmation

To commit the current candidate configuration but require an explicit confirmation for the commit to become permanent, emit the empty <confirmed/> tag enclosed in <commit-configuration> and <rpc> tag elements. If the commit is not confirmed within a certain amount of time (10 minutes by default), the JUNOScript server automatically rolls back to the previously committed configuration. To specify a different number of minutes for the rollback deadline, also emit the <confirm-timeout> tag element enclosing a positive integer value.

The JUNOScript server encloses its response in <rpc-reply>, <commit-results>, and <routing-engine> tag elements. If the commit operation succeeds, the <routing-engine> tag element encloses the <commit-success/> tag and the <name> tag element, which reports the name of the Routing Engine on which the commit operation succeeded (either re0 or re1). If the commit operation fails, an <xnm:error> tag element encloses tag elements that describe the error. The most common causes of failure are semantic or syntactic errors in the candidate configuration.

The <confirmed/> tag is useful for verifying that a configuration change works correctly and does not prevent management access to the router. If the change prevents access or causes other errors, the automatic rollback to the previous configuration restores access after the rollback deadline passes.

To delay the rollback to a time later than the current rollback deadline, emit the <confirmed/> tag again (enclosed in a <commit-configuration> tag element) before the deadline passes. Optionally, include the <confirm-timeout> tag element to specify how long to delay the next rollback; omit that tag element to delay the rollback by the default 10 minutes. The client application can delay the rollback indefinitely by emitting the <confirmed/> tag repeatedly in this way.

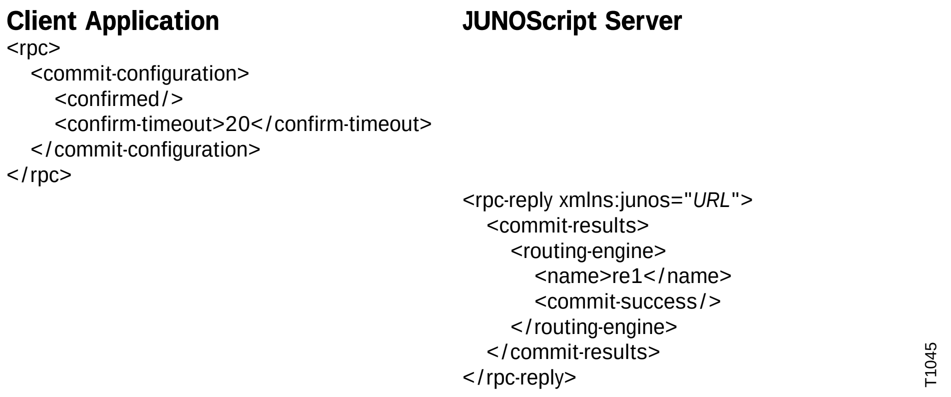
To cancel the rollback completely (and commit a configuration permanently), emit one of the following tag sequences before the rollback deadline passes:

- The empty <commit-configuration/> tag enclosed in an <rpc> tag element. The rollback is cancelled and the current candidate configuration is committed immediately, as described in “Commit the Candidate Configuration” on page 72. If the candidate configuration is still the same as the temporarily committed configuration, this effectively recommits the temporarily committed configuration.

- The <commit-synchronize/> tag enclosed in <commit-configuration> and <rpc> tag elements. The rollback is cancelled and the current candidate configuration is checked and committed immediately on both Routing Engines, as described in “Commit and Synchronize the Configuration on Both Routing Engines” on page 73. If a confirmed commit operation has been performed on both Routing Engines, then emitting the <commit-synchronize/> tag cancels the rollback on both.

- The <commit-at> tag element enclosed in <commit-configuration> and <rpc> tag elements. The rollback is cancelled and the configuration is checked immediately for syntactic correctness, then committed at the scheduled time, as described in “Commit the Configuration at a Specified Time” on page 74.

The following example illustrates the tag sequence when a client application commits the candidate configuration on Routing Engine 1 with a rollback deadline of 20 minutes:



Client Application

```
<rpc>
    <commit-configuration>
        <confirmed/>
        <confirm-timeout>20</confirm-timeout>
    </commit-configuration>
</rpc>
```

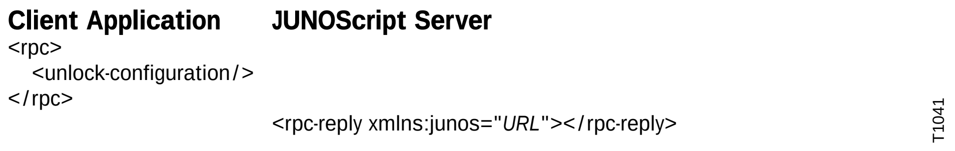
JUNOScript Server

```
<rpc-reply xmlns:junos="URL">
    <commit-results>
        <routing-engine>
            <name>re1</name>
            <commit-success/>
        </routing-engine>
    </commit-results>
</rpc-reply>
```

Unlock the Candidate Configuration

To unlock the candidate configuration after changing it, committing it, or both, emit the empty <unlock-configuration/> tag enclosed in an <rpc> tag element. Other applications and users cannot change the candidate configuration until the client application releases the lock. To confirm that it has successfully unlocked the configuration, the JUNOScript server returns an opening <rpc-reply> and closing </rpc-reply> tag with nothing between them. If it cannot unlock the configuration, it returns an <xnm:error> tag element instead.

The following example illustrates the tag sequence with which the client application unlocks the configuration:



Client Application

```
<rpc>
    <unlock-configuration/>
</rpc>
```

JUNOScript Server

```
<rpc-reply xmlns:junos="URL"></rpc-reply>
```

Part 3
Write JUNOScript Client Applications

Chapter 5 Write Perl Client Applications

Juniper Networks provides a Perl module, called JUNOS, to help you more quickly and easily develop custom Perl scripts for configuring and monitoring routers. The module implements an object, called JUNOS::Device, that client applications can use to communicate with the JUNOScript server on a router. Accompanying the JUNOS module are several sample Perl scripts, which illustrate how to use the module in scripts that perform various functions.

This chapter discusses the following topics:

Overview of the JUNOS Module and Sample Scripts

The JUNOScript Perl distribution uses the same directory structure for Perl modules as the Comprehensive Perl Archive Network (http://www.cpan.org). This includes a lib directory for the JUNOS module and its supporting files, and an examples directory for the sample scripts.

The JUNOS module implements an object (JUNOS::Device) that client applications can use to communicate with a JUNOScript server. All of the sample scripts use the object.

The sample scripts illustrate how to perform the following functions:

- diagnose_bgp.pl—Illustrates how to write scripts to monitor router status and diagnose problems. The sample script extracts and displays information about a router's unestablished Border Gateway Protocol (BGP) peers from the full set of BGP configuration data. The script is provided in the examples/diagnose_bgp directory in the JUNOScript Perl distribution.

- get_chassis_inventory.pl—Illustrates how to use one of the predefined Perl JUNOScript queries to request information from a router. The sample script invokes the get_chassis_inventory query with the detail option to request the same information as the JUNOScript <get-chassis-inventory><detail/></get-chassis-inventory> tag sequence and the JUNOS command-line interface (CLI) show chassis hardware detail command. The script is provided in the examples/get_chassis_inventory directory in the JUNOScript Perl distribution. For a list of all Perl queries available in this release of JUNOScript, see Table 6 on page 107.

- load_configuration.pl—Illustrates how to change router configuration by loading a file that contains configuration data formatted with JUNOScript tag elements. The distribution includes two sample configuration files, set_login_user_foo.xml and set_login_class_bar.xml; however, you can specify another JUNOScript configuration file on the command line. The script is provided in the examples/load_configuration directory in the JUNOScript Perl distribution.

The following sample scripts are used together to illustrate how to store and retrieve JUNOScript (or any Extensible Markup Language [XML]) data in a relational database. While these scripts create and manipulate MySQL tables, the data manipulation techniques that they illustrate apply to any relational database. The scripts are provided in the examples/RDB directory in the JUNOScript Perl distribution:

- get_config.pl—Illustrates how to retrieve router configuration information.
- make_tables.pl—Generates a set of Structured Query Language (SQL) statements for creating relational database tables.
- pop_tables.pl—Populates existing relational database tables with data extracted from a specified XML file.
- unpop_tables.pl—Transforms data stored in a relational database table into XML and writes it to a file.

For instructions on running the scripts, see the README or README.html file included in the JUNOScript Perl distribution.

Download the JUNOS Module and Sample Scripts

To download, uncompress, and unpack the compressed tar-format file that contains the JUNOS module and sample scripts, perform the following steps:

1. Access the Juniper Networks Customer Support Center Web page at http://www.juniper.net/support.
2. Click on the link labeled JUNOScript API Software.
3. On the JUNOScript API Software Download page, click on the link for the appropriate JUNOS Internet software release.
4. To download the JUNOScript API Perl client package and the prerequisites package, click on the links for the packages that support the appropriate access protocols. Customers in the United States and Canada can download the package that supports all access protocols (the domestic package), or the package that supports clear-text and telnet only (the export package). Customers in other countries can download the package that supports clear-text and telnet only.

   Note It is assumed that the machine on which you store and run the Perl client software is a regular computer instead of a Juniper Networks router.

5. Optionally, download the package containing document type definitions (DTDs).

6. Change to the directory where you want to create a subdirectory that contains the JUNOS Perl module and sample scripts:

   ```
   % cd perl-parent-directory
   ```

7. Issue the following command to uncompress and unpack the package downloaded in Step 4:

   - On FreeBSD and Linux systems:

     ```
     % tar zxf junoscript-perl-release-type.tar.gz
     ```

   - On Solaris systems:

     ```
     % gzip -dc junoscript-perl-release-type.tar.gz | tar xf
     ```

   where *release* is the JUNOS release code (such as 5.6R1.1) and *type* is domestic or export. The command creates a directory called junoscript-perl-*release-type* and writes the contents of the tar file to it.

8. See the junoscript-perl-*release-type*/README file for instructions on unpacking and installing the Perl prerequisite modules, creating a Makefile, and installing and testing the JUNOS module.

Tutorial: Writing Perl Client Applications

This tutorial explains how to write a Perl client application that requests operational or configuration information from the JUNOScript server or loads configuration information onto a router. The following sections use the sample scripts included in the JUNOScript Perl distribution as examples:

- Import Perl Modules and Declare Constants on page 84
- Connect to the JUNOScript Server on page 84
- Submit a Request to the JUNOScript Server on page 91
- Parse and Format the Response from the JUNOScript Server on page 100
- Close the Connection to the JUNOScript Server on page 106

Import Perl Modules and Declare Constants

Include the following statements at the start of the application. The first statement imports the functions provided by the JUNOS::Device object, which the application uses to connect to the JUNOScript server on a router. The second statement provides error checking and enforces Perl coding practices such as declaration of variables before use.

```
use JUNOS::Device;
use strict;
```

Include other statements to import other Perl modules as appropriate for your application. For example, several of the sample scripts import the following standard Perl modules, which include functions that handle input from the command line:

- Getopt::Std—Includes functions for reading in keyed options from the command line.
- Term::ReadKey—Includes functions for controlling terminal modes, for example suppressing onscreen echo of a typed string such as a password.
- File::Basename—Includes functions for processing filenames.

If the application uses constants, declare their values at this point. For example, the sample diagnose_bgp.pl script includes the following statements to declare constants for formatting output:

```
use constant OUTPUT_FORMAT => "%-20s%-8s%-8s%-11s%-14s%s\n";
use constant OUTPUT_TITLE => "\n======================== BGP PROBLEM SUMMARY
========================\n\n";
use constant OUTPUT_ENDING =>
"\n==============================================================
===\n\n";
```

The load_configuration.pl script includes the following statements to declare constants for reporting return codes and the status of the configuration database:

```
use constant REPORT_SUCCESS => 1;
use constant REPORT_FAILURE => 0;
use constant STATE_CONNECTED => 1;
use constant STATE_LOCKED => 2;
use constant STATE_CONFIG_LOADED => 3;
```

Connect to the JUNOScript Server

The following sections explain how to use the JUNOS::Device object to connect to the JUNOScript server on a router:

- Satisfy Protocol Prerequisites on page 85
- Group Requests on page 85
- Obtain and Record Parameters Required by the JUNOS::Device Object on page 85
- Obtain Application-Specific Parameters on page 88
- Convert Disallowed Characters on page 89
- Establish the Connection on page 91

Satisfy Protocol Prerequisites

The JUNOScript server supports several access protocols, listed in "Supported Access Protocols" on page 14. For each connection to the JUNOScript server on a router, the application must specify the protocol it is using. Using secure shell (ssh) or Secure Sockets Layer (SSL) is recommended because they provide greater security by encrypting all information before transmission across the network.

Before your application can run, you must satisfy the prerequisites for the protocol it uses. For some protocols this involves activating configuration statements on the router, creating encryption keys, or installing additional software on the router or the machine where the application runs. For instructions, see "Prerequisites for Establishing a Connection" on page 15.

Group Requests

Establishing a connection to the JUNOScript server on a router is one of the more time- and resource-intensive functions performed by an application. If the application sends multiple requests to a router, it makes sense to send all of them within the context of one connection. If your application sends the same requests to multiple routers, you can structure the script to iterate through either the set of routers or the set of requests. Keep in mind, however, that your application can effectively send only one request to one JUNOScript server at a time. This is because the JUNOS::Device object does not return control to the application until it receives the closing </rpc-reply> tag that represents the end of the JUNOScript server's response to the current request.

Obtain and Record Parameters Required by the JUNOS::Device Object

The JUNOS::Device object takes the following required parameters, specified as keys in a Perl hash:

- The access protocol to use when communicating with the JUNOScript server (key name: access). For a list of the acceptable values, see "Supported Access Protocols" on page 14. Before the application runs, satisfy the protocol-specific prerequisites described in "Prerequisites for Establishing a Connection" on page 15.

- The name of the router to which to connect (key name: hostname). For best results, specify either a fully-qualified hostname or an IP address.

- The username of the JUNOS login account under which to establish the JUNOScript connection and issue requests (key name: login). The account must already exist on the specified router and have the JUNOS permission bits necessary for making the requests invoked by the application.

- The password for the JUNOS login account (key name: password).

The sample scripts record the parameters in a Perl hash called %deviceinfo, declared as follows:

```
my %deviceinfo = (
        access => $access,
        login => $login,
        password => $password,
        hostname => $hostname,
    );
```

The sample scripts obtain the parameters from options entered on the command line by a user. Your application can also obtain values for the parameters from a file or database, or you can hardcode one or more of the parameters into the application code if they are constant.

Example: Collect Parameters Interactively

Each sample script obtains the parameters required by the JUNOS::Device object from command-line options provided by the user who invokes the script. The script records the options in a Perl hash called %opt, using the getopts function defined in the Getopt::Std Perl module to read the options from the command line. (Scripts used in production environments probably do not obtain parameters interactively, so this section is important mostly for understanding the sample scripts.)

In the following example from the get_chassis_inventory.pl script, the first parameter to the getopts function defines the acceptable options, which vary depending on the application. A colon after the option letter indicates that it takes an argument. The second parameter, \%opt, specifies that the values are recorded in the %opt hash. If the user does not provide at least one option, provides an invalid option, or provides the -h option, the script invokes the output_usage subroutine, which prints a usage message to the screen:

```
my %opt;
getopts('l:p:dm:x:o:h', \%opt) || output_usage();
output_usage() if $opt{h};
```

The following code defines the output_usage subroutine for the get_chassis_inventory.pl script. The contents of the my $usage definition and the Where and Options sections are specific to the script, and differ for each application.

```
sub output_usage
{
    my $usage = "Usage: $0 [options] <target>

Where:

  <target>   The hostname of the target router.

Options:

  -l <login>         A login name accepted by the target router.
  -p <password>   The password for the login name.
  -m <access>      Access method.  It can be clear-text, ssl, ssh or telnet.  Default: telnet.
  -x <format>       The name of the XSL file to display the response.
                          Default: xsl/chassis_inventory_csv.xsl
  -o <filename>    File to which to write output, instead of standard output.
  -d                     Turn on debugging.\n\n";

    die $usage;
}
```

The get_chassis_inventory.pl script includes the following code to obtain values from the command line for the four parameters required by the JUNOS::Device object. A detailed discussion of the various functional units follows the complete code sample.

```
my $hostname = shift || output_usage();

my $access = $opt{m} || "telnet";
use constant VALID_ACCESSES => "telnet|ssh|clear-text|ssl";
output_usage() unless (VALID_ACCESSES =~ /$access/);

my $login = "";
if ($opt{l}) {
    $login = $opt{l};
} else {
    print "login: ";
    $login = ReadLine 0;
    chomp $login;
}

my $password = "";
if ($opt{p}) {
    $password = $opt{p};
} else {
    print "password: ";
    ReadMode 'noecho';
    $password = ReadLine 0;
    chomp $password;
    ReadMode 'normal';
    print "\n";
}
```

In the first line of the preceding code sample, the script uses the Perl shift function to read the hostname from the end of the command line. If the hostname is missing, the script invokes the output_usage subroutine to print the usage message, which specifies that a hostname is required:

```
my $hostname = shift || output_usage();
```

The script next determines which access protocol to use, setting the $access variable to the value of the -m command-line option or to the value telnet if the -m option is not provided. If the specified value does not match one of the four values defined by the VALID_ACCESSES constant, the script invokes the output_usage subroutine:

```
my $access = $opt{m} || "telnet";
use constant VALID_ACCESSES => "telnet|ssh|clear-text|ssl";
output_usage() unless ($access =~ /VALID_ACCESSES/);
```

The script then determines the JUNOS login account name, setting the $login variable to the value of the -l command-line option. If the option is not provided, the script prompts for it and uses the ReadLine function (defined in the standard Perl Term::ReadKey module) to read it from the command line:

```
my $login = "";
if ($opt{l}) {
    $login = $opt{l};
} else {
    print "login: ";
    $login = ReadLine 0;
    chomp $login;
}
```

The script finally determines the password for the JUNOS account, setting the $password variable to the value of the -p command-line option. If the option is not provided, the script prompts for it. It uses the ReadMode function (defined in the standard Perl Term::ReadKey module) twice: first to prevent the password from echoing visibly on the screen and then to return the shell to normal (echo) mode after it reads the password:

```
my $password = "";
if ($opt{p}) {
    $password = $opt{p};
} else {
    print "password: ";
    ReadMode 'noecho';
    $password = ReadLine 0;
    chomp $password;
    ReadMode 'normal';
    print "\n";
}
```

Obtain Application-Specific Parameters

In addition to the parameters required by the JUNOS::Device object, applications might need to define other parameters, such as the name of the file to which to write the data returned by the JUNOScript server in response to a request, or the name of the Extensible Stylesheet Transformation Language (XSLT) file to use for transforming the data.

As with the parameters required by the JUNOS::Device object, your application can hardcode the values in the application code, obtain them from a file, or obtain them interactively. The sample scripts obtain values for these parameters from command-line options in the same manner as they obtain the parameters required by the JUNOS::Device object (discussed in "Obtain and Record Parameters Required by the JUNOS::Device Object" on page 85). Several examples follow.

The following line enables a debugging trace if the user includes the -d command-line option. It invokes the JUNOS::Trace::init routine defined in the JUNOS::Trace module, which is already imported with the JUNOS::Device object.

```
JUNOS::Trace::init(1) if $opt{d};
```

The following line sets the $outputfile variable to the value specified by the -o command-line option. It names the local file to which the JUNOScript server's response is written. If the -o option is not provided, the variable is set to the empty string.

```
my $outputfile = $opt{o} || "";
```

The following code from the diagnose_bgp.pl script defines which XSLT file to use to transform the JUNOScript server's response. The first line sets the $xslfile variable to the value specified by the -x command-line option. If the option is not provided, the script uses the text.xsl file supplied with the script, which transforms the data to ASCII text. The if statement verifies that the specified XSLT file exists; the script terminates if it does not.

```
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";
}
```

The following code from the load_configuration.pl script defines whether to merge, replace, or overwrite the new configuration data into the configuration database (for more information about these operations, see “Change the Candidate Configuration” on page 61). The first two lines set the $load_action variable to the value of the -a command-line option, or to the default value merge if the option is not provided. If the specified value does not match one of the four defined in the third line, the script invokes the output_usage subroutine.

```
# The default action for load_configuration is 'merge'
my $load_action = "merge";
$load_action = $opt{a} if $opt{a};
use constant VALID_ACTIONS => "merge|replace|override";
output_usage() unless ($load_action =~ /VALID_ACTIONS/);
```

Convert Disallowed Characters

Scripts that handle configuration data usually accept and output the data either as JUNOScript tag elements or as ASCII-formatted statements like those used in the JUNOS CLI. As described in “Predefined Entity References” on page 12, certain characters cannot appear in their regular form in an XML document. These characters include the apostrophe (‘), the ampersand (&), the greater-than (>) and less-than (<) symbols, and the quotation mark ("). Because these characters might appear in ASCII-formatted configuration statements, the script must convert the characters to the corresponding predefined entity references.

The load_configuration.pl script uses the get_escaped_text subroutine to substitute predefined entity references for disallowed characters (the get_configuration.pl script includes similar code). The script first defines the mappings between the disallowed characters and predefined entity references, and sets the variable $char_class to a regular expression that contains all of the entity references, as follows:

```
my %escape_symbols = (
    qq(") => '&quot;',
    qq(>) => '&gt;',
    qq(<) => '&lt;',
    qq(') => '&apos;',
    qq(&) => '&amp;'
);

my $char_class = join ("|", map { "($_)" } keys %escape_symbols);
```

The following code defines the get_escaped_text subroutine for the load_configuration.pl script. A detailed discussion of the subsections in the routine follows the complete code sample.

```
sub get_escaped_text
{
    my $input_file = shift;
    my $input_string = "";

    open(FH, $input_file) or return undef;

    while(<FH>) {
        my $line = $_;
        $line =~ s/<configuration-text>//g;
        $line =~ s/<\/configuration-text>//g;
        $line =~ s/($char_class)/$escape_symbols{$1}/ge;
        $input_string .= $line;
    }

    return "<configuration-text>$input_string</configuration-text>";
}
```

The first subsection of the preceding code sample reads in a file containing ASCII-formatted configuration statements:

```
sub get_escaped_text
{
    my $input_file = shift;
    my $input_string = "";

    open(FH, $input_file) or return undef;
```

In the next subsection, the subroutine temporarily discards the lines that contain the opening <get-configuration> and closing </get-configuration> tags, then replaces the disallowed characters on each remaining line with predefined entity references and appends the line to the $input_string variable:

```
    while(<FH>) {
        my $line = $_;
        $line =~ s/<configuration-text>//g;
        $line =~ s/<\/configuration-text>//g;
        $line =~ s/($char_class)/$escape_symbols{$1}/ge;
        $input_string .= $line;
    }
```

The subroutine concludes by replacing the opening <get-configuration> and closing </get-configuration> tags, and returning the converted set of statements:

```
    return "<configuration-text>$input_string</configuration-text>";
}
```

Establish the Connection

After obtaining values for the parameters required for the JUNOS::Device object (see "Obtain and Record Parameters Required by the JUNOS::Device Object" on page 85), each sample script records them in the %deviceinfo hash:

```
my %deviceinfo = (
    access => $access,
    login => $login,
    password => $password,
    hostname => $hostname,
);
```

The script then invokes the JUNOScript-specific new subroutine to create a JUNOS::Device object and establish a connection to the specified router. If the connection attempt fails (as tested by the ref operator), the script exits.

```
my $jnx = new JUNOS::Device(%deviceinfo);
unless ( ref $jnx ) {
    die "ERROR: $deviceinfo{hostname}: failed to connect.\n";
```

Submit a Request to the JUNOScript Server

After establishing a connection to a JUNOScript server (see "Establish the Connection" on page 91), your application can submit one or more requests by invoking the Perl methods that are supported in the version of the JUNOScript API used by the application:

- Each version of JUNOS software supports a set of methods that correspond to JUNOS CLI operational mode commands (later releases generally support more methods). For a list of the operational methods supported in the current version, see "Summary of Mappings Between Perl Queries and JUNOScript Tag Elements" on page 106 and the files stored in the lib/JUNOS/*release* directory of the JUNOScript Perl distribution (*release* is the version code such as 5.6R1 for the initial release of JUNOS 5.6). The files have names in the format *package*_methods.pl, where *package* is a JUNOS software package.

- The set of methods that correspond to operations on JUNOS configuration objects is defined in the file lib/JUNOS/Methods.pm in the JUNOScript Perl distribution. For more information about configuration operations, see "Change the Candidate Configuration" on page 61 and the chapter about session control tags in the *JUNOScript API Reference*.

See the following sections for more information:

- Provide Method Options or Attributes on page 92
- Submit a Request on page 94
- Example: Get an Inventory of Hardware Components on page 95
- Example: Load Configuration Statements on page 96

Provide Method Options or Attributes

Many Perl methods have one or more options or attributes. The following list describes the notation used to define a method's options in the lib/JUNOS/Methods.pm and lib/JUNOS/*release*/*package*_methods.pl files, and the notation that an application uses when invoking the method:

- A method without options is defined as $NO_ARGS, as in the following entry for the get_system_uptime_information method:

  ```
  ## Method : <get-system-uptime-information>
  ## Returns: <system-uptime-information>
  ## Command: "show system uptime"
  get_system_uptime_information => $NO_ARGS,
  ```

  To invoke a method without options, follow the method name with an empty set of parentheses as in the following example:

  ```
  $jnx->get_system_uptime_information();
  ```

- A fixed-form option is defined as type $TOGGLE. In the following example, the get_software_information method takes two fixed-form options, brief and detail:

  ```
  ## Method : <get-software-information>
  ## Returns: <software-information>
  ## Command: "show version"
  get_software_information => {
      brief => $TOGGLE,
      detail => $TOGGLE,
  },
  ```

  To include a fixed-form option when invoking a method, set it to the value 1 (one) as in the following example:

  ```
  $jnx->get_software_information(brief => 1);
  ```

- An option with a variable value is defined as type $STRING. In the following example, the get_cos_drop_profile_information method takes the profile_name argument:

  ```
  ## Method : <get-cos-drop-profile-information>
  ## Returns: <cos-drop-profile-information>
  ## Command: "show class-of-service drop-profile"
  get_cos_drop_profile_information => {
      profile_name => $STRING,
  };
  ```

  To include a variable value when invoking a method, enclose the value in single quotes as in the following example:

  ```
  $jnx->get_cos_drop_profile_information(profile_name => 'user-drop-profile');
  ```

- An attribute is defined as type $ATTRIBUTE. In the following example, the lock_configuration method takes the rollback attribute:

  ```
  lock_configuration => {
      rollback => $ATTRIBUTE
  },
  ```

  To include a numerical attribute value when invoking a method, set it to the appropriate value. The following example rolls the candidate configuration back to the previous configuration that has an index of 2:

  ```
  $jnx->load_configuration(rollback => 2);
  ```

  To include a string attribute value when invoking a method, enclose the value in single quotes as in the following example:

  ```
  $jnx->get_configuration(format => ‘text’);
  ```

- A set of configuration statements or corresponding tag elements is defined as type $DOM. In the following example, the get_configuration method takes a set of configuration statements (along with two attributes):

  ```
  get_configuration => {
      configuration => $DOM,
      format => $ATTRIBUTE,
      database => $ATTRIBUTE,
  },
  ```

  To include a set of configuration statements when invoking a method, provide a parsed set of statements or tags. The following example refers to a set of JUNOScript configuration tags in the config-input.xml file. For further discussion, see “Example: Load Configuration Statements” on page 96.

  ```
  my $parser = new XML::DOM::Parser;
  $jnx->load_configuration(
          format => ‘xml’,
          action => ‘merge’,
          configuration => $parser->parsefile(config-input.xml)
  );
  ```

A method can have a combination of fixed-form options, options with variable values, attributes, and a set of configuration statements. For example, the get_route_forwarding_table method has four fixed-form options and five options with variable values:

```
## Method : <get-forwarding-table-information>
## Returns: <forwarding-table-information>
## Command: "show route forwarding-table"
get_forwarding_table_information => {
    detail => $TOGGLE,
    extensive => $TOGGLE,
    multicast => $TOGGLE,
    family => $STRING,
    vpn => $STRING,
    summary => $TOGGLE,
    matching => $STRING,
    destination => $STRING,
    label => $STRING,
},
```

Submit a Request

The following is the recommended way to send a request to the JUNOScript server. It assumes that the $jnx variable was previously defined to be a JUNOS::Device object, as discussed in "Establish the Connection" on page 91.

The following code sends a request to the JUNOScript server and handles error conditions. A detailed discussion of the functional subsections follows the complete code sample.

```
my %arguments = ();
%arguments = (argument1 => value1,
    argument2 => value2, ...);
    argument3 => value3,
    ...);

my $res = $jnx->method(%args);

unless ( ref $res ) {
    $jnx->request_end_session();
    $jnx->disconnect();
    print "ERROR: Could not send request to $hostname\n";
}

my $err = $res->getFirstError();
if ($err) {
    $jnx->request_end_session();
    $jnx->disconnect();
    print "ERROR: Error for $hostname: " . $err->{message} . "\n";
}
```

The first subsection of the preceding code sample creates a hash called %arguments to define values for a method's options or attributes. For each argument, the application uses the notation described in "Provide Method Options or Attributes" on page 92.

```
my %arguments = ();
%arguments = (argument1 => value1,
    argument2 => value2, ...);
    argument3 => value3,
    ...);
```

The application then invokes the method, defining the $res variable to point to the JUNOS::Response object that the JUNOScript server returns in response to the request (the object is defined in the lib/JUNOS/Response.pm file in the JUNOScript Perl distribution):

```
my $res = $jnx->method(%args);
```

If the attempt to send the request failed, the application prints an error message and closes the connection:

```
unless ( ref $res ) {
   $jnx->request_end_session();
   $jnx->disconnect();
   print "ERROR: Could not send request to $hostname\n";
}
```

If there was an error in the JUNOScript server's response, the application prints an error message and closes the connection. The getFirstError function is defined in the JUNOS::Response module (lib/JUNOS/Response.pm) in the JUNOScript Perl distribution.

```
my $err = $res->getFirstError();
if ($err) {
   $jnx->request_end_session();
   $jnx->disconnect();
   print "ERROR: Error for $hostname: " . $err->{message} . "\n";
}
```

Example: Get an Inventory of Hardware Components

The get_chassis_inventory.pl script retrieves and displays a detailed inventory of the hardware components installed in a router. It is equivalent to issuing the show chassis hardware detail command.

After establishing a connection to the JUNOScript server, the script defines get_chassis_inventory as the request to send and includes the detail argument:

```
my $query = "get_chassis_inventory";
my %queryargs = ( detail => 1 );
```

The script sends the query and assigns the results to the $res variable. It performs two tests on the results, and prints an error message if it cannot send the request or if errors occurred when executing it. If no errors occurred, the script uses XSLT to transform the results. For more information, see "Parse and Format an Operational Response" on page 101.

```
my $res = $jnx->$query( %queryargs );
unless ( ref $res ) {
   die "ERROR: $deviceinfo{hostname}: failed to execute command $query.\n";
}
my $err = $res->getFirstError();
if ($err) {
   print STDERR "ERROR: $deviceinfo{'hostname'} - ", $err->{message}, "\n";
} else {
   ... code to process results with XSLT ...
}
```

Example: Load Configuration Statements

The load_configuration.pl script loads configuration statements onto a router. It uses the basic structure for sending requests described in “Submit a Request” on page 94, but also defines a graceful_shutdown subroutine that handles errors in a slightly more elaborate manner than that described in “Submit a Request” on page 94. The following sections describe the different functions that the script performs:

- Handle Error Conditions on page 96
- Lock the Configuration on page 97
- Read In and Parse the Configuration Data on page 98
- Load the Configuration Data on page 99
- Commit the Configuration on page 100

Handle Error Conditions

The graceful_shutdown subroutine in the load_configuration.pl script handles errors in a slightly more elaborate manner that the generic structure described in “Submit a Request” on page 94. It employs the following additional constants:

```
use constant REPORT_SUCCESS => 1;
use constant REPORT_FAILURE => 0;
use constant STATE_CONNECTED => 1;
use constant STATE_LOCKED => 2;
use constant STATE_CONFIG_LOADED => 3;
```

The first two if statements in the subroutine refer to the STATE_CONFIG_LOADED and STATE_LOCKED conditions, which apply specifically to loading a configuration in the load_configuration.pl script. The if statement for STATE_CONNECTED is similar to the error checking described in "Submit a Request" on page 94. The eval statement used in each case ensures that any errors that occur during execution of the enclosed function call are trapped so that failure of the function call does not cause the script to exit.

```
sub graceful_shutdown
{
    my ($jnx, $req, $state, $success) = @_;

    if ($state >= STATE_CONFIG_LOADED) {
        print "Rolling back configuration ...\n";
        eval {
            $jnx->load_configuration(rollback => 0);
        };
    }

    if ($state >= STATE_LOCKED) {
        print "Unlocking configuration database ...\n";
        eval {
            $jnx->unlock_configuration();
        };
    }

    if ($state >= STATE_CONNECTED) {
        print "Disconnecting from the router ...\n";
        eval {
            $jnx->request_end_session();
            $jnx->disconnect();
        };
    }

    if ($success) {
        die "REQUEST $req SUCCEEDED\n";
    } else {
        die "REQUEST $req FAILED\n";
    };
}
```

Lock the Configuration

The main section of the load_configuration.pl script begins by establishing a connection to a JUNOScript server, as described in "Establish the Connection" on page 91. It then invokes the lock_configuration method to lock the configuration database. In case of error, the script invokes the graceful_shutdown subroutine described in "Handle Error Conditions" on page 96.

```
print "Locking configuration database ...\n";
my $res = $jnx->lock_configuration();
my $err = $res->getFirstError();
if ($err) {
    print "ERROR: $deviceinfo{hostname}: failed to lock configuration.  Reason:
            $err->{message}.\n";
    graceful_shutdown($jnx, $xmlfile, STATE_CONNECTED, REPORT_FAILURE);
}
```

Read In and Parse the Configuration Data

In the following code sample, the load_configuration.pl script then reads in and parses a file that contains JUNOScript configuration tag elements or ASCII-formatted statements. The name of the file was previously obtained from the command line and assigned to the $xmlfile variable. A detailed discussion of the functional subsections follows the complete code sample.

```
print "Loading configuration from $xmlfile ...\n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

my $parser = new XML::DOM::Parser;
...
my $doc;
if ($opt{t}) {
    my $xmlstring = get_escaped_text($xmlfile);
    $doc = $parser->parsestring($xmlstring) if $xmlstring;

} else {
    $doc = $parser->parsefile($xmlfile);
}

unless ( ref $doc ) {
    print "ERROR: Cannot parse $xmlfile, check to make sure the XML data is well-formed\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
```

The first subsection of the preceding code sample verifies the existence of the file containing configuration data. The name of the file was previously obtained from the command line and assigned to the $xmlfile variable. If the file does not exist, the script invokes the graceful_shutdown subroutine:

```
print "Loading configuration from $xmlfile ...\n";
if (! -f $xmlfile) {
    print "ERROR: Cannot load configuration in $xmlfile\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
```

If the -t command-line option was included when the load_configuration.pl script was invoked, the file referenced by the $xmlfile variable should contain ASCII-formatted configuration statements like those returned by the CLI configuration-mode show command. The script invokes the get_escaped_text subroutine described in "Convert Disallowed Characters" on page 89, assigning the result to the $xmlstring variable. The script invokes the parsestring function to transform the data in the file into the proper format for loading into the configuration hierarchy, and assigns the result to the $doc variable. The parsestring function is defined in the XML::DOM::Parser module, and the first line in the following sample code instantiates the module as an object, setting the $parser variable to refer to it:

```
my $parser = new XML::DOM::Parser;
...
my $doc;
if ($opt{t}) {
    my $xmlstring = get_escaped_text($xmlfile);
    $doc = $parser->parsestring($xmlstring) if $xmlstring;
```

If the file contains JUNOScript configuration tags instead, the script invokes the parsefile function (also defined in the XML::DOM::Parser module) on the file:

```
} else {
   $doc = $parser->parsefile($xmlfile);
}
```

If the parser cannot transform the file, the script invokes the graceful_shutdown subroutine described in “Handle Error Conditions” on page 96:

```
unless ( ref $doc ) {
   print "ERROR: Cannot parse $xmlfile, check to make sure the XML data is well-formed\n";
   graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}
```

Load the Configuration Data

The script now invokes the load_configuration method to load the configuration onto the router. It places the statement inside an eval block to ensure that the graceful_shutdown subroutine is invoked if the response from the JUNOScript server has errors.

```
eval {
$res = $jnx->load_configuration(
   format => $config_format,
   action => $load_action,
   configuration => $doc);
};
if ($@) {
   print "ERROR: Failed to load the configuration from $xmlfile.   Reason: $@\n";
   graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
   exit(1);
}
```

The variables used to define the method’s three arguments were set at previous points in the application file:

- The $config_format variable was previously set to xml unless the -t command-line option is included:

  ```
  my $config_format = "xml";
  $config_format = "text" if $opt{t};
  ```

- The $load_action variable was previously set to merge unless the -a command-line option is included. The final two lines verify that the specified value is one of the acceptable choices:

  ```
  my $load_action = "merge";
  $load_action = $opt{a} if $opt{a};
  use constant VALID_ACTIONS => "merge|replace|override";
  output_usage() unless ( $load_action =~ /VALID_ACTIONS/);
  ```

- The $doc variable was set to the output from the parsestring or parsefile function (defined in the XML::DOM::Parser module), as described in “Read In and Parse the Configuration Data” on page 98.

The script performs two additional checks for errors and invokes the graceful_shutdown subroutine in either case:

```
unless ( ref $res ) {
    print "ERROR: Failed to load the configuration from $xmlfile\n";
    graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_FAILURE);
}

$err = $res->getFirstError();
if ($err) {
    print "ERROR: Failed to load the configuration.  Reason: $err->{message}\n";
    graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
}
```

Commit the Configuration

If there are no errors, the script invokes the commit_configuration method (defined in the file lib/JUNOS/Methods.pm in the JUNOScript Perl distribution):

```
print "Committing configuration from $xmlfile ...\n";
$res = $jnx->commit_configuration();
$err = $res->getFirstError();
if ($err) {
    print "ERROR: Failed to commit configuration.  Reason: $err->{message}.\n";
    graceful_shutdown($jnx, $xmlfile, STATE_CONFIG_LOADED, REPORT_FAILURE);
}
```

Parse and Format the Response from the JUNOScript Server

As the last step in sending a request, the application verifies that there are no errors with the response from the JUNOScript server (see “Submit a Request” on page 94). It can then write the response to a file, to the screen, or both. If the response is for an operational query, the application usually uses XSLT to transform the output into a more readable format, such as HTML or formatted ASCII. If the response consists of configuration data, the application can store it as XML (the JUNOScript tag elements generated by default from the JUNOScript server) or transform it into formatted ASCII.

The following sections discuss parsing and formatting options:

- Parse and Format an Operational Response on page 101
- Parse and Output Configuration Data on page 103

Parse and Format an Operational Response

The following code sample from the diagnose_bgp.pl and get_chassis_inventory.pl scripts uses XSLT to transform an operational response from the JUNOScript server into a more readable format. A detailed discussion of the functional subsections follows the complete code sample.

```
my $outputfile = $opt{o} || "";

my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";

my $xmlfile = "$deviceinfo{hostname}.xml";
$res->printToFile($xmlfile);

my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile, "$xslfile.tmp");

if ($nm) {
    print "Transforming $xmlfile with $xslfile...\n" if $outputfile;
    my $command = "xsltproc $nm $deviceinfo{hostname}.xml";

    $command .= "> $outputfile" if $outputfile;
    system($command);
    print "Done\n" if $outputfile;
    print "See $outputfile\n" if $outputfile;
}

else {
    print STDERR "ERROR: Invalid XSL file $xslfile\n";
}
```

The first line of the preceding code sample illustrates how the scripts read the -o option from the command line to obtain the name of file into which to write the results of the XSLT transformation:

```
my $outputfile = $opt{o} || "";
```

From the -x command-line option, the scripts obtain the name of the XSLT file to use, setting a default value if the option is not provided. The scripts exit if the specified file does not exist. The following example is from the diagnose_bgp.pl script:

```
my $xslfile = $opt{x} || "xsl/text.xsl";
if ($xslfile && ! -f $xslfile) {
    die "ERROR: XSLT file $xslfile does not exist";
```

For examples of XSLT files, see the following directories in the JUNOScript Perl distribution:

- The examples/diagnose_bpg/xsl directory contains XSLT files for the diagnose_bpg.pl script: dhtml.xsl generates dynamic HTML, html.xsl generates HTML, and text.xsl generates ASCII.
- The examples/get_chassis_inventory/xsl directory contains XSLT files for the get_chassis_inventory.pl script: chassis_inventory_csv.xsl generates a list of comma-separated values, chassis_inventory_html.xsl generates HTML, and chassis_inventory_xml.xsl generates XML.

The actual parsing operation begins by setting the variable $xmlfile to a filename of the form *router-name*.xml and invoking the printToFile function to write the JUNOScript server's response into the file (the printToFile function is defined in the XML::DOM::Parser module):

```
my $xmlfile = "$deviceinfo{hostname}.xml";
$res->printToFile($xmlfile);
```

The next line invokes the translateXSLtoRelease function (defined in the JUNOS::Response module) to alter one of the namespace definitions in the XSLT file. This is necessary because the XSLT 1.0 specification requires that every XSLT file define a specific value for each default namespace used in the data being transformed. The xmlns attribute on a JUNOScript operational response tag element includes a code representing the JUNOS version, such as 5.6R1 for the initial version of JUNOS Release 5.6. Because the same XSLT file can be applied to operational response tag elements from routers running different versions of JUNOS, the XSLT file cannot predefine an xmlns namespace value that matches all versions. The translateXSLtoRelease function alters the namespace definition in the XSLT file identified by the $xslfile variable to match the value in the JUNOScript server's response. It assigns the resulting XSLT file to the $nm variable.

```
my $nm = $res->translateXSLtoRelease('xmlns:lc', $xslfile, "$xslfile.tmp");
```

After verifying that the translateXSLtoRelease function succeeded, the function builds a command string and assigns it to the $command variable. The first part of the command string invokes the xsltproc command and specifies the names of the XSLT and configuration data files ($nm and $deviceinfo{hostname}.xml):

```
if ($nm) {
   print "Transforming $xmlfile with $xslfile...\n" if $outputfile;
   my $command = "xsltproc $nm $deviceinfo{hostname}.xml";
```

If the $outputfile variable is defined (the file for storing the result of the XSLT transformation exists), the script appends a string to the $command variable to write the results of the xsltproc command to the file. (If the file does not exist, the script writes the results to standard out [stdout].) The script then invokes the system function to execute the command string and prints status messages to stdout.

```
   $command .= "> $outputfile" if $outputfile;
   system($command);
   print "Done\n" if $outputfile;
   print "See $outputfile\n" if $outputfile;
}
```

If the translateXSLtoRelease function fails (the if ($nm) expression evaluates to "false"), the script prints an error:

```
else {
   print STDERR "ERROR: Invalid XSL file $xslfile\n";
}
```

Parse and Output Configuration Data

The get_config.pl script uses the outconfig subroutine to write the configuration data obtained from the JUNOScript server to a file as either JUNOScript tags or formatted ASCII.

The outconfig subroutine takes four parameters. Three must have defined values: the directory in which to store the output file, the router hostname, and the XML DOM tree (the configuration data) returned by the JUNOScript server. The fourth parameter indicates whether to output the configuration as formatted ASCII, and has a null value if the output should be JUNOScript tag elements. In the following code sample, the script obtains values for the four parameters and passes them to the subroutine. A detailed discussion of each line follows the complete code sample.

```
my(%opt,$login,$password);

getopts('l:p:dm:hit', \%opt) || output_usage();
output_usage() if $opt{h};

my $basepath = shift || output_usage;

my $hostname = shift || output_usage;

my $config = getconfig( $hostname, $jnx, $opt{t} );

outconfig( $basepath, $hostname, $config, $opt{t} );
```

In the first lines of the preceding sample code, the get_config.pl script uses the following statements to obtain values for the four parameters to the outconfig subroutine:

- If the user provides the -t option on the command line, the getopts subroutine records it in the %opt hash. The value keyed to $opt{t} is passed as the fourth parameter to the outconfig subroutine. (For more information about reading options from the command line, see “Example: Collect Parameters Interactively” on page 86.)

  ```
  getopts('l:p:dm:hit', \%opt) || output_usage();
  ```

- The following line reads the first element of the command line that is not an option preceded by a hyphen. It assigns the value to the $basepath variable, defining the name of the directory in which to store the file containing the output from the outconfig subroutine. The variable value is passed as the first parameter to the outconfig subroutine.

  ```
  my $basepath = shift || output_usage;
  ```

- The following line reads the next element on the command line. It assigns the value to the $hostname variable, defining the router hostname. The variable value is passed as the second parameter to the outconfig subroutine.

  ```
  my $hostname = shift || output_usage;
  ```

- The following line invokes the getconfig subroutine to obtain configuration data from the JUNOScript server on the specified router, assigning the resulting XML DOM tree to the $config variable. The variable value is passed as the third parameter to the outconfig subroutine.

  ```
  my $config = getconfig( $hostname, $jnx, $opt{t} );
  ```

The following code sample invokes and defines the outconfig subroutine. A detailed discussion of each functional subsection in the subroutine follows the complete code sample.

```
outconfig( $basepath, $hostname, $config, $opt{t} );

sub outconfig( $$$$ ) {
    my $leader = shift;
    my $hostname = shift;
    my $config = shift;
    my $text_mode = shift;
    my $trailer = "xmlconfig";
    my $filename = $leader . "/" . $hostname . "." . $trailer;

    print "# storing configuration for $hostname as $filename\n";

    my $config_node;
    my $top_tag = "configuration";
    $top_tag .= "-text" if $text_mode;
    if ($config->getTagName() eq $top_tag) {
        $config_node = $config;
    } else {
        print "# unknown response component ", $config->getTagName(), "\n";
    }

    if ( $config_node && $config_node ne "" ) {
        if ( open OUTPUTFILE, ">$filename" )    {
            if (!$text_mode) {
                print OUTPUTFILE "<?xml version=\"1.0\"?>\n";
                print OUTPUTFILE $config_node->toString(), "\n";
            } else {
                my $buf = $config_node->getFirstChild()->toString();
                $buf =~ s/($char_class)/$escapes{$1}/ge;
                print OUTPUTFILE "$buf\n";
            }
            close OUTPUTFILE;
        }
        else {
            print "ERROR: could not open output file $filename\n";
        }
    }
    else {
        print "ERROR: empty configuration data for $hostname\n";
    }
}
```

The first lines of the outconfig subroutine read in the four parameters passed in when the subroutine is invoked, assigning each to a local variable:

```
outconfig( $basepath, $hostname, $config, $opt{t} );

sub outconfig( $$$$ ) {
    my $leader = shift;
    my $hostname = shift;
    my $config = shift;
    my $text_mode = shift;
```

The subroutine constructs the name of the file to which to write the subroutine's output and assigns the name to the $filename variable. The filename is constructed from the first two parameters (the directory name and hostname) and the $trailer variable, resulting in a name of the form *directory-name*/*hostname*.xmlconfig:

```
my $trailer = "xmlconfig";
my $filename = $leader . "/" . $hostname . "." . $trailer;

print "# storing configuration for $hostname as $filename\n";
```

The subroutine checks that the first tag in the XML DOM tree correctly indicates the type of configuration data in the file. If the user included the -t option on the command line, the first tag should be <configuration-text> because the file contains formatted ASCII configuration statements; otherwise, the first tag should be <configuration> because the file contains JUNOScript tag elements. The subroutine sets the $top_tag variable to the appropriate value depending on the value of the $text_mode variable (which takes its value from opt{t}, passed as the fourth parameter to the subroutine). The subroutine invokes the getTagName function (defined in the XML::DOM::Element module) to retrieve the name of the first tag in the input file, and compares the name to the value of the $top_tag variable. If the comparison succeeds, the XML DOM tree is assigned to the $config_node variable. Otherwise, the subroutine prints an error message because the XML DOM tree is not valid configuration data.

```
my $config_node;
my $top_tag = "configuration";
$top_tag .= "-text" if $text_mode;
if ($config->getTagName() eq $top_tag) {
    $config_node = $config;
} else {
    print "# unknown response component ", $config->getTagName(), "\n";
}
```

The subroutine then uses several nested if statements. The first if statement verifies that the XML DOM tree exists and contains data:

```
if ( $config_node && $config_node ne "" ) {
    ... actions if XML DOM tree contains data ...
}
else {
    print "ERROR: empty configuration data for $hostname\n";
}
```

If the XML DOM tree contains data, the subroutine verifies that the output file can be opened for writing:

```
if ( open OUTPUTFILE, ">$filename" )    {
    ... actions if output file is writable ...
}
 else {
    print "ERROR: could not open output file $filename\n";
}
```

If the output file can be opened for writing, the script writes the configuration data into it. If the user requested JUNOScript tag elements (the $text_mode variable does not have a value because the user did not include the -t option on the command line), the script writes the string <?xml version=1.0?> as the first line in the output file, then invokes the toString function (defined in the XML::DOM module) to write each JUNOScript tag element in the XML DOM tree on a line in the output file:

```
if (!$text_mode) {
    print OUTPUTFILE "<?xml version=\"1.0\"?>\n";
    print OUTPUTFILE $config_node->toString(), "\n";
```

If the user requested formatted ASCII, the script invokes the getFirstChild and toString functions (defined in the XML::DOM module) to write the content of each tag on its own line in the output file. The script substitutes predefined entity references for disallowed characters (which are defined in the %escapes hash), writes the output to the output file, and closes the output file. (For information about defining the %escapes hash to contain the set of disallowed characters, see “Convert Disallowed Characters” on page 89.)

```
} else {
    my $buf = $config_node->getFirstChild()->toString();
    $buf =~ s/($char_class)/$escapes{$1}/ge;
    print OUTPUTFILE "$buf\n";
  }
  close OUTPUTFILE;
```

Close the Connection to the JUNOScript Server

To end the JUNOScript session and close the connection to the router, each sample script invokes the request_end_session and disconnect methods. Several of the scripts do this in standalone statements:

```
$jnx->request_end_session();
$jnx->disconnect();
```

The load_configuration.pl script invokes the graceful_shutdown subroutine instead (for more information, see “Handle Error Conditions” on page 96):

```
graceful_shutdown($jnx, $xmlfile, STATE_LOCKED, REPORT_SUCCESS);
```

Summary of Mappings Between Perl Queries and JUNOScript Tag Elements

The sample scripts described in “Overview of the JUNOS Module and Sample Scripts” on page 81 invoke only a small number of the predefined JUNOScript Perl queries that client applications can use. Table 6 maps all of the Perl queries available in the current version of JUNOScript to the corresponding JUNOScript response tag element and JUNOS CLI command. Each query has the same name as the corresponding JUNOScript request tag element (to derive the name of the request tag element, replace each underscore with a hyphen and enclose the string in angle brackets).

For more information about JUNOScript request and response tag elements, see the *JUNOScript API Reference*.

Table 6: Mapping of Perl / Java Methods to JUNOScript Tags

Method	Response Tag	CLI Command
clear_arp_table	<clear-arp-table-results>	clear arp
clear_helper_statistics_information	NONE	clear helper statistics
clear_ipv6_nd_information	<ipv6-modify-nd>	clear ipv6 neighbors
file_compare	NONE	file compare
file_copy	NONE	file copy
file_delete	NONE	file delete
file_list	<directory-list>	file list
file_rename	NONE	file rename
file_show	<file-content>	file show
get_accounting_profile_information	<accounting-profile-information>	show accounting profile
get_accounting_record_information	<accounting-record-information>	show accounting records
get_alarm_information	<alarm-information>	show chassis alarms
get_arp_table_information	<arp-table-information>	show arp
get_bgp_group_information	<bgp-group-information>	show bgp group
get_bgp_neighbor_information	<bgp-information>	show bgp neighbor
get_bgp_summary_information	<bgp-information>	show bgp summary
get_chassis_inventory	<chassis-inventory>	show chassis hardware
get_cos_classifier_information	<cos-classifier-information>	show class-of-service classifier
get_cos_classifier_table_information	<cos-classifier-table-information>	show class-of-service forwarding-table classifier
get_cos_classifier_table_map_information	<cos-classifier-table-map-information>	show class-of-service forwarding-table classifier mapping
get_cos_code_point_map_information	<cos-code-point-map-information>	show class-of-service code-point-aliases
get_cos_drop_profile_information	<cos-drop-profile-information>	show class-of-service drop-profile
get_cos_fabric_scheduler_map_information	<cos-fabric-scheduler-map-information>	show class-of-service fabric scheduler-map
get_cos_forwarding_class_information	<cos-forwarding-class-information>	show class-of-service forwarding-class
get_cos_fwtab_fabric_scheduler_map_information	<cos-fwtab-fabric-scheduler-map-information>	show class-of-service forwarding-table fabric scheduler-map
get_cos_information	<cos-information>	show class-of-service
get_cos_interface_map_information	<cos-interface-information>	show class-of-service interface
get_cos_red_information	<cos-red-information>	show class-of-service forwarding-table drop-profile
get_cos_rewrite_information	<cos-rewrite-information>	show class-of-service rewrite-rule
get_cos_rewrite_table_information	<cos-rewrite-table-information>	show class-of-service forwarding-table rewrite-rule
get_cos_rewrite_table_map_information	<cos-rewrite-table-map-information>	show class-of-service forwarding-table rewrite-rule mapping
get_cos_scheduler_map_information	<cos-scheduler-map-information>	show class-of-service scheduler-map
get_cos_scheduler_map_table_information	<cos-scheduler-map-table-information>	show class-of-service forwarding-table scheduler-map
get_cos_table_information	<cos-table-information>	show class-of-service forwarding-table
get_destination_class_statistics	<destination-class-statistics>	show interfaces destination-class
get_environment_information	<environment-information>	show chassis environment
get_fabric_queue_information	<fabric-queue-information>	show class-of-service fabric statistics
get_feb_information	<scb-information>	show chassis feb

Method	Response Tag	CLI Command
get_firewall_information	<firewall-information>	show firewall
get_firewall_log_information	<firewall-log-information>	show firewall log
get_firewall_prefix_action_information	<firewall-prefix-action-information>	show firewall prefix-action-stats
get_firmware_information	<firmware-information>	show chassis firmware
get_forwarding_table_information	<forwarding-table-information>	show route forwarding-table
get_fpc_information	<fpc-information>	show chassis fpc
get_ggsn_apn_statistics_information	<apn-statistics-information>	show services ggsn statistics apn
get_ggsn_gtp_prime_statistics_information	<gtp-prime-statistics-information>	show services ggsn statistics gtp-prime
get_ggsn_gtp_statistics_information	<gtp-statistics-information>	show services ggsn statistics gtp
get_ggsn_imsi_trace	<call-trace-information>	show services ggsn trace imsi
get_ggsn_imsi_user_information	<mobile-user-information>	show services ggsn statistics imsi
get_ggsn_interface_information	<ggsn-interface-information>	show services ggsn status
get_ggsn_msisdn_trace	<call-trace-information>	show services ggsn trace msisdn
get_ggsn_sgsn_statistics_information	<sgsn-statistics-information>	show services ggsn statistics sgsn
get_ggsn_statistics	<ggsn-statistics>	show services ggsn statistics
get_ggsn_trace	<call-trace-information>	show services ggsn trace all
get_helper_statistics_information	<helper-statistics-information>	show helper statistics
get_ike_security_associations_information	<ike-security-associations-information>	show ike security-associations
get_instance_information	<instance-information>	show route instance
get_interface_filter_information	<interface-filter-information>	show interfaces filters
get_interface_information	<interface-information>	show interfaces
get_interface_policer_information	<interface-policer-information>	show interfaces policers
get_interface_queue_information	<interface-information>	show interfaces queue
get_ipv6_nd_information	<ipv6-nd-information>	show ipv6 neighbors
get_ipv6_ra_information	<ipv6-ra-information>	show ipv6 router-advertisement
get_isis_adjacency_information	<isis-adjacency-information>	show isis adjacency
get_isis_database_information	<isis-database-information>	show isis database
get_isis_hostname_information	<isis-hostname-information>	show isis hostname
get_isis_interface_information	<isis-interface-information>	show isis interface
get_isis_route_information	<isis-route-information>	show isis route
get_isis_spf_information	<isis-spf-information>	show isis spf
get_isis_statistics_information	<isis-statistics-information>	show isis statistics
get_l2ckt_connection_information	<l2ckt-connection-information>	show l2circuit connections
get_l2vpn_connection_information	<l2vpn-connection-information>	show l2vpn connections
get_ldp_database_information	<ldp-database-information>	show ldp database
get_ldp_interface_information	<ldp-interface-information>	show ldp interface
get_ldp_neighbor_information	<ldp-neighbor-information>	show ldp neighbor
get_ldp_path_information	<ldp-path-information>	show ldp path
get_ldp_route_information	<ldp-route-information>	show ldp route
get_ldp_session_information	<ldp-session-information>	show ldp session
get_ldp_statistics_information	<ldp-statistics-information>	show ldp statistics
get_ldp_traffic_statistics_information	<ldp-traffic-statistics-information>	show ldp traffic-statistics
get_lm_information	<lm-information>	show link-management

Method	Response Tag	CLI Command
get_lm_peer_information	<lm-peer-information>	show link-management peer
get_lm_routing_information	<lm-information>	show link-management routing
get_lm_routing_peer_information	<lm-peer-information>	show link-management routing peer
get_lm_routing_te_link_information	<lm-te-link-information>	show link-management routing te-link
get_lm_te_link_information	<lm-te-link-information>	show link-management te-link
get_mpls_admin_group_information	<mpls-admin-group-information>	show mpls admin-groups
get_mpls_cspf_information	<mpls-cspf-information>	show mpls cspf
get_mpls_interface_information	<mpls-interface-information>	show mpls interface
get_mpls_lsp_information	<mpls-lsp-information>	show mpls lsp
get_mpls_path_information	<mpls-path-information>	show mpls path
get_ospf_database_information	<ospf-database-information>	show ospf database
get_ospf_interface_information	<ospf-interface-information>	show ospf interface
get_ospf_io_statistics_information	<ospf-io-statistics-information>	show ospf io-statistics
get_ospf_log_information	<ospf-log-information>	show ospf log
get_ospf_neighbor_information	<ospf-neighbor-information>	show ospf neighbor
get_ospf_route_information	<ospf-route-information>	show ospf route
get_ospf_statistics_information	<ospf-statistics-information>	show ospf statistics
get_ospf3_database_information	<ospf3-database-information>	show ospf3 database
get_ospf3_interface_information	<ospf3-interface-information>	show ospf3 interface
get_ospf3_io_statistics_information	<ospf3-io-statistics-information>	show ospf3 io-statistics
get_ospf3_log_information	<ospf3-log-information>	show ospf3 log
get_ospf3_neighbor_information	<ospf3-neighbor-information>	show ospf3 neighbor
get_ospf3_route_information	<ospf3-route-information>	show ospf3 route
get_ospf3_statistics_information	<ospf3-statistics-information>	show ospf3 statistics
get_passive_monitoring_error_information	<passive-monitoring-error-information>	show passive-monitoring error
get_passive_monitoring_flow_information	<passive-monitoring-flow-information>	show passive-monitoring flow
get_passive_monitoring_information	<passive-monitoring-information>	show passive-monitoring
get_passive_monitoring_memory_information	<passive-monitoring-memory-information>	show passive-monitoring memory
get_passive_monitoring_status_information	<passive-monitoring-status-information>	show passive-monitoring status
get_passive_monitoring_usage_information	<passive-monitoring-usage-information>	show passive-monitoring usage
get_pic_detail	<pic-information>	show chassis pic
get_pic_information	<fpc-information>	show chassis fpc pic-status
get_rmon_alarm_information	<rmon-alarm-information>	show snmp rmon alarms
get_rmon_event_information	<rmon-event-information>	show snmp rmon events
get_rmon_information	<rmon-information>	show snmp rmon
get_route_engine_information	<route-engine-information>	show chassis routing-engine
get_route_information	<route-information>	show route
get_route_summary_information	<route-summary-information>	show route summary
get_rsvp_interface_information	<rsvp-interface-information>	show rsvp interface
get_rsvp_neighbor_information	<rsvp-neighbor-information>	show rsvp neighbor
get_rsvp_session_information	<rsvp-session-information>	show rsvp session
get_rsvp_statistics_information	<rsvp-statistics-information>	show rsvp statistics
get_rsvp_version_information	<rsvp-version-information>	show rsvp version

Method	Response Tag	CLI Command
get_rtexport_instance_information	<rtexport-instance-information>	show route export instance
get_rtexport_table_information	<rtexport-table-information>	show route export
get_rtexport_target_information	<rtexport-target-information>	show route export vrf-target
get_scb_information	<scb-information>	show chassis scb
get_security_associations_information	<security-associations-information>	show ipsec security-associations
get_sfm_information	<scb-information>	show chassis sfm
get_snmp_information	<snmp-statistics>	show snmp statistics
get_software_information	<software-information>	show version
get_source_class_statistics	<source-class-statistics>	show interfaces source-class
get_spmb_information	<spmb-information>	show chassis spmb
get_spmb_sib_information	<spmb-sib-information>	show chassis spmb sibs
get_ssb_information	<scb-information>	show chassis ssb
get_syslog_tag_information	<syslog-tag-information>	help syslog
get_system_storage	<system-storage-information>	show system storage
get_system_uptime_information	<system-uptime-information>	show system uptime
get_system_users_information	<system-users-information>	show system users
get_ted_database_information	<ted-database-information>	show ted database
get_ted_link_information	<ted-link-information>	show ted link
get_ted_protocol_information	<ted-protocol-information>	show ted protocol
request_end_session	<end-session>	quit
request_ggsn_restart_interface	<interface-action-results>	request services ggsn restart interface
request_ggsn_restart_node	<node-action-results>	request services ggsn restart node
request_ggsn_start_imsi_trace	NONE	request services ggsn trace start imsi
request_ggsn_start_msisdn_trace	NONE	request services ggsn trace start msisdn
request_ggsn_stop_imsi_trace	NONE	request services ggsn trace stop imsi
request_ggsn_stop_msisdn_trace	NONE	request services ggsn trace stop msisdn
request_ggsn_stop_trace_activity	NONE	request services ggsn trace stop all
request_ggsn_terminate_context	<pdp-context-deletion-results>	request services ggsn pdp terminate context
request_ggsn_terminate_contexts_apn	<apn-pdp-context-deletion-results>	request services ggsn pdp terminate apn
request_halt	NONE	request system halt
request_package_add	NONE	request system software add
request_package_delete	NONE	request system software delete
request_package_validate	NONE	request system software validate
request_reboot	NONE	request system reboot
request_snapshot	NONE	request system snapshot

Chapter 6

Write a C Client Application

The following example illustrates how a client application written in C can use the secure shell (ssh) or telnet protocol to establish a JUNOScript connection and session. In the line that begins with the string execlp, the client application invokes the ssh command. (Substitute the telnet command if appropriate.) The *router* argument to the execlp routine specifies the hostname or IP address of the JUNOScript server. The junoscript argument is the command that converts the connection to a JUNOScript session.

For more information about JUNOScript sessions, see “Start, Control, and End a JUNOScript Session” on page 14.

```
int ipipes[ 2 ], opipes[ 2 ];
pid_t pid;
int rc;
char buf[ BUFSIZ ];

if (pipe(ipipes) <0 || pipe(opipes) <0)
    err(1, "pipe failed");

pid = fork();
if (pid <0)
    err(1, "fork failed");

if (pid == 0) {
    dup2(opipes[ 0 ], STDIN_FILENO);
    dup2(ipipes[ 1 ], STDOUT_FILENO);
    dup2(ipipes[ 1 ], STDERR_FILENO);
    close(ipipes[ 0 ]); /* close read end of pipe */
    close(ipipes[ 1 ]); /* close write end of pipe */
    close(opipes[ 0 ]); /* close read end of pipe */
    close(opipes[ 1 ]); /* close write end of pipe */

    execlp("ssh", "ssh", "-x", router, "junoscript", NULL);
    err (1, "unable to execute: ssh %s junoscript," router);
}

close(ipipes[ 1 ]); /* close write end of pipe */
close(opipes[ 0 ]); /* close read end of pipe */

if (write(opipes[ 1 ], initial_handshake, strlen(initial_handshake)) <0 )
    err(1, "writing initial handshake failed");

rc=read(ipipes[ 0 ], buf, sizeof(buf));
if (rc <0)
    err(1, "read initial handshake failed");
```

Part 4
Index

Index